SECOND EDITION Covers Python 3



THE Quick Python Book

First edition by Daryl K. Harms
Kenneth M. McDonald

Vernon L. Ceder

MANNING

The Quick Python Book
Second Edition

The Quick Python Book

SECOND EDITION

VERNON L. CEDER

FIRST EDITION BY DARYL K. HARMS
KENNETH M. McDONALD

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact:

Special Sales Department
Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830
Email: orders@manning.com

Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Development editor: Tara Walsh
Copyeditor: Linda Recktenwald
Typesetter: Marija Tudor
Cover designer: Leslie Haimes

ISBN 9781935182207

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 15 14 13 12 11 10 09

brief contents

contents

preface

I've been coding in Python for a number of years, longer than any other language I've ever used. I use Python for system administration, for web applications, for database management, and sometimes just to help myself think clearly.

To be honest, I'm sometimes a little surprised that Python has worn so well. Based on my earlier experience, I would have expected that by now some other language would have come along that was faster, cooler, sexier, whatever. Indeed, other languages have come along, but none that helped me do what I needed to do quite as effectively as Python. In fact, the more I use Python and the more I understand it, the more I feel the quality of my programming improve and mature.

This is a second edition, and my mantra in updating has been, "If it ain't broke, don't fix it." Much of the content has been freshened for Python 3 but is largely as written in the first edition. Of course, the world of Python has changed since Python 1.5, so in several places I've had to make significant changes or add new material. On those occasions I've done my best to make the new material compatible with the clear and low-key style of the original.

For me, the aim of this book is to share the positive experiences I've gotten from coding in Python by introducing people to Python 3, the latest and, in my opinion, the best version of Python to date. May your journey be as satisfying as mine has been.

acknowledgments

I want to thank David Fugate of LaunchBooks for getting me into this book in the first place and for all of the support and advice he has provided over the years. I can't imagine having a better agent and friend. I also need to thank Michael Stephens of Manning for pushing the idea of doing a second edition of this book and supporting me in my efforts to make it as good as the first. Also at Manning, many thanks to every person who worked on this project, with special thanks to Marjan Bace for his support, Tara Walsh for guidance in the development phases, Mary Piergies for getting the book (and me) through the production process, Linda Recktenwald for her patience in copy editing, and Tiffany Taylor for proofreading. I also owe a huge debt to Will Kahn-Greene for all of the astute advice he gave both as a technical reviewer and in doing the technical proofing. Thanks, Will, you saved me from myself more times than I can count. Likewise, hearty thanks to the many reviewers whose insights and feedback were of immense help: Nick Lo, Michele Galli, Andy Dingley, Mohamed Lamkadem, Robby O'Connor, Amos Bannister, Joshua Miller, Christian Marquardt, Andrew Rhine, Anthony Briggs, Carlton Gibson, Craig Smith, Daniel McKinnon, David McWhirter, Edmon Begoli, Elliot Winard, Horaci Macias, Massimo Perga, Munch Paulson, Nathan R. Yergler, Rick Wagner, Sumit Pal, and Tyson S. Maxwell.

Because this is a second edition, I have to thank the authors of the first edition, Daryl Harms and Kenneth MacDonald, for two things: first, for writing a book so sound that it has remained in print well beyond the average lifespan of most tech books, and second, for being otherwise occupied, thereby giving me a chance to update it. I hope this version carries on the successful and long-lived tradition of the first.

Thanks to my canine associates, Molly, Riker, and Aeryn, who got fewer walks, training sessions, and games of ball than they should have but still curled up beside my chair and kept me company and helped me keep my sense of perspective as I worked. You'll get those walks now, guys, I promise.

Most important, thanks to my wife, Becky, who both encouraged me to take on this project and had to put up with the most in the course of its completion—particularly an often-grumpy and preoccupied spouse. I really couldn't have done it without you.

about this book

Who should read this book

This book is intended for people who already have experience in one or more programming languages and want to learn the basics of Python 3 as quickly and directly as possible. Although some basic concepts are covered, there's no attempt to teach basic programming skills in this book, and the basic concepts of flow control, OOP, file access, exception handling, and the like are assumed. This book may also be of use to users of earlier versions of Python who want a concise reference for Python 3.

How to use this book

Part 1 introduces Python and explains how to download and install it on your system. It also includes a very general survey of the language, which will be most useful for experienced programmers looking for a high-level view of Python.

Part 2 is the heart of the book. It covers the ingredients necessary for obtaining a working knowledge of Python as a general-purpose programming language. The chapters are designed to allow readers who are beginning to learn Python to work their way through sequentially, picking up knowledge of the key points of the language. These chapters also contain some more advanced sections, allowing you to return to find in one place all the necessary information about a construct or topic.

Part 3 introduces advanced language features of Python, elements of the language that aren't essential to its use but that can certainly be a great help to a serious Python programmer.

Part 4 describes more advanced or specialized topics that are beyond the strict syntax of the language. You may read these chapters or not, depending on your needs.

A suggested plan if you're new to Python is to start by reading chapter 3 to obtain an overall perspective and then work through the chapters in part 2 that are applicable. Enter in the interactive examples as they are introduced. This will immediately reinforce the concepts. You can also easily go beyond the examples in the text to answer questions about anything that may be unclear. This has the potential to amplify the speed of your learning and the level of your comprehension. If you aren't familiar with OOP or don't need it for your application, skip most of chapter 15. If you aren't interested in developing a GUI, skip chapter 16.

Those familiar with Python should also start with chapter 3. It will be a good review and will introduce differences between Python 3 and what may be more familiar. It's a reasonable test of whether you're ready to move on to the advanced chapters in parts 3 and 4 of this book.

It's possible that some readers, although new to Python, will have enough experience with other programming languages to be able to pick up the bulk of what they need to get going from chapter 3 and by browsing the Python standard library modules listed in chapter 23 and the *Python Library Reference* in the Python documentation.

Roadmap

Chapter 1 discusses the strengths and weaknesses of Python and shows why Python 3 is a good choice of programming language for many situations.

Chapter 2 covers downloading, installing, and starting up the Python interpreter and IDLE, its integrated development environment.

Chapter 3 is a short overview of the Python language. It provides a basic idea of the philosophy, syntax, semantics, and capabilities of the language.

Chapter 4 starts with the basics of Python. It introduces Python variables, expressions, strings, and numbers. It also introduces Python's block-structured syntax.

Chapters 5, 6, and 7 describe the five powerful built-in Python data types: lists, tuples, sets, strings, and dictionaries.

Chapter 8 introduces Python's control flow syntax and use (loops and `if-else` statements).

Chapter 9 describes function definition in Python along with its flexible parameter-passing capabilities.

Chapter 10 describes Python modules. They provide an easy mechanism for segmenting the program namespace.

Chapter 11 covers creating standalone Python programs, or scripts, and running them on Windows, Mac OS X, and Linux platforms. The chapter also covers the support available for command-line options, arguments, and I/O redirection.

Chapter 12 describes how to work and navigate through the files and directories of the filesystem. It shows how to write code that's as independent as possible of the actual operating system you're working on.

Chapter 13 introduces the mechanisms for reading and writing files in Python. These include the basic capability to read and write strings (or byte streams), the mechanism available for reading binary records, and the ability to read and write arbitrary Python objects.

Chapter 14 discusses the use of exceptions, the error-handling mechanism used by Python. It doesn't assume that you have any prior knowledge of exceptions, although if you've previously used them in C++ or Java, you'll find them familiar.

Chapter 15 introduces Python's support for writing object-oriented programs.

Chapter 16 focuses on the available Tkinter interface and ends with an introduction to some of the other options available for developing GUIs.

Chapter 17 discusses the regular-expression capabilities available for Python.

Chapter 18 introduces the package concept in Python for structuring the code of large projects.

Chapter 19 covers the simple mechanisms available to dynamically discover and work with data types in Python.

Chapter 20 introduces more advanced OOP techniques, including the use of Python's special method-attributes mechanism, metaclasses, and abstract base classes.

Chapter 21 covers two strategies that Python offers for testing your code: doctests and unit testing.

Chapter 22 surveys the process, issues, and tools involved in porting code from earlier versions of Python to Python 3.

Chapter 23 is a brief survey of the standard library and also includes a discussion of where to find other modules and how to install them.

Chapter 24 is a brief introduction to using Python for database and web programming. A small web application is developed to illustrate the principles involved.

The appendix contains a comprehensive guide to obtaining and accessing Python's full documentation, the Pythonic style guide, PEP 8, and "The Zen of Python," a slightly wry summary of the philosophy behind Python.

Code conventions

The code samples in this book, and their output, appear in a `fixed-width font` and are often accompanied by annotations. The code samples are deliberately kept as simple as possible, because they aren't intended to be reusable parts that can plugged into your code. Instead, the code samples are stripped down so that you can focus on the principle being illustrated.

In keeping with the idea of simplicity, the code examples are presented as interactive shell sessions where possible; you should enter and experiment with these samples as much as you can. In interactive code samples, the commands that need to be entered are on lines that begin with the `>>>` prompt, and the visible results of that code (if any) are on the line below.

In some cases a longer code sample is needed, and these are identified in the text as file listings. You should save these as files with names matching those used in the text and run them as standalone scripts.

Source code downloads

The source code for the samples in this book is available from the publisher's website at www.manning.com/TheQuickPythonBookSecondEdition.

System requirements

The samples and code in this book have been written with Windows (XP through Windows 7), Mac OS X, and Linux in mind. Because Python is a cross-platform language, they *should* work on other platforms for the most part, except for platform-specific issues, like the handling of files, paths, and GUIs.

Software requirements

This book is based on Python 3.1, and all examples should work on any subsequent version of Python 3. The examples also work on Python 3.0, but I strongly recommend using 3.1—there are no advantages to the earlier version, and 3.1 has several subtle improvements. Note that Python 3 is *required* and that an earlier version of Python will not work with the code in this book.

Author online

The purchase of *The Quick Python Book, Second Edition* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/TheQuickPythonBookSecondEdition. This page provides information about how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It's not a commitment to any specific amount of participation on the part of the author, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking him some challenging questions, lest his interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the author

Second edition author Vern Ceder has been programming in various languages for over 20 years and has been a Linux system administrator since 2000. He started using Python for a variety of projects in 2001 and is director of technology at the Canterbury School in Fort Wayne, Indiana, where he teaches Python to high school students and

teachers and gives talks to whomever will listen on Python and the benefits of teaching programming in schools. An advocate for open software and open content, Vern is a principal organizer of the Fort Wayne Linux Users Group.

About the cover illustration

The illustration on the cover of *The Quick Python Book, Second Edition* is taken from a late 18th century edition of Sylvain Maréchal's four-volume compendium of regional dress customs published in France. Each illustration is finely drawn and colored by hand. The rich variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by what they were wearing.

Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

Part 1

Starting out

This section will tell you a little bit about Python, its strengths and weaknesses, and why you should consider learning Python 3. You'll also see how to install Python on Windows, Mac OS X, and Linux platforms and how to write a simple program.

Finally, chapter 3 is a quick, high-level survey of Python's syntax and features. If you're looking for the quickest possible introduction to Python, read chapter 3.

1 About Python

This chapter covers

- Why use Python?
- What Python does well
- What Python doesn't do as well
- Why learn Python 3?

Read this chapter if you want to know how Python compares to other languages and its place in the grand scheme of things. Skip this chapter if you want to start learning Python right away. The information in this chapter is a valid part of this book—but it's certainly not necessary for programming with Python.

1.1 Why should I use Python?

Hundreds of programming languages are available today, from mature languages like C and C++, to newer entries like Ruby, C#, and Lua, to enterprise juggernauts like Java. Choosing a language to learn is difficult. Although no one language is the right choice for every possible situation, I think that Python is a good choice for a large number of programming problems, and it's also a good choice if you're learning to program. Hundreds of thousands of programmers around the world use Python, and the number grows every year.

Python continues to attract new users for a variety of reasons. It's a true cross-platform language, running equally well on Windows, Linux/UNIX, and Macintosh platforms, as well as others, ranging from supercomputers to cell phones. It can be used to develop small applications and rapid prototypes, but it scales well to permit development of large programs. It comes with a powerful and easy-to-use graphical user interface (GUI) toolkit, web programming libraries, and more. *And* it's free.

1.2 What Python does well

Python is a modern programming language developed by Guido van Rossum in the 1990s (and named after a famous comedic troupe). Although Python isn't perfect for every application, its strengths make it a good choice for many situations.

1.2.1 Python is easy to use

Programmers familiar with traditional languages will find it easy to learn Python. All of the familiar constructs such as loops, conditional statements, arrays, and so forth are included, but many are easier to use in Python. Here are a few of the reasons why:

- *Types are associated with objects, not variables.* A variable can be assigned a value of any type, and a list can contain objects of many different types. This also means that type casting usually isn't necessary, and your code isn't locked into the straitjacket of predeclared types.
- *Python typically operates at a much higher level of abstraction.* This is partly the result of the way the language is built and partly the result of an extensive standard code library that comes with the Python distribution. A program to download a web page can be written in two or three lines!
- *Syntax rules are very simple.* Although becoming an expert Pythonista takes time and effort, even beginners can absorb enough Python syntax to write useful code quickly.

Python is well suited for rapid application development. It isn't unusual for coding an application in Python to take one-fifth the time it would if coded in C or Java and to take as little as one-fifth the number of lines of the equivalent C program. This depends on the particular application, of course; for a numerical algorithm performing mostly integer arithmetic in `for` loops, there would be much less of a productivity gain. For the average application, the productivity gain can be significant.

1.2.2 Python is expressive

Python is a very expressive language. *Expressive* in this context means that a single line of Python code can do more than a single line of code in most other languages. The advantages of a more expressive language are obvious: the fewer lines of code you have to write, the faster you can complete the project. Not only that, but the fewer lines of code there are, the easier the program will be to maintain and debug.

To get an idea of how Python's expressiveness can simplify code, let's consider swapping the values of two variables, `var1` and `var2`. In a language like Java, this requires three lines of code and an extra variable:

```
int temp = var1;
var1 = var2;
var2 = temp;
```

The variable `temp` is needed to save the value of `var1` when `var2` is put into it, and then that saved value is put into `var2`. The process isn't terribly complex, but reading those three lines and understanding that a swap has taken place takes a certain amount of overhead, even for experienced coders.

In contrast, Python lets you make the same swap in one line and in a way that makes it obvious that a swap of values has occurred:

```
var2, var1 = var1, var2
```

Of course, this is a very simple example, but you find the same advantages throughout the language.

1.2.3 Python is readable

Another advantage of Python is that it's easy to read. You might think that a programming language needs to be read only by a computer, but humans have to read your code as well—whoever debugs your code (quite possibly you), whoever maintains your code (could be you again), and whoever might want to modify your code in the future. In all of those situations, the easier the code is to read and understand, the better it is.

The easier code is to understand, the easier it is to debug, maintain, and modify. Python's main advantage in this department is its use of indentation. Unlike most languages, Python *insists* that blocks of code be indented. Although this strikes some as odd, it has the benefit that your code is always formatted in a very easy-to-read style.

Following are two short programs, one written in Perl and one in Python. Both take two equal-sized lists of numbers and return the pairwise sum of those lists. I think the Python code is more readable than the Perl code; it's visually cleaner and contains fewer inscrutable symbols:

```
# Perl version.
sub pairwise_sum {
    my($arg1, $arg2) = @_;
    my(@result) = ();
    @list1 = @$arg1;
    @list2 = @$arg2;
    for($i=0; $i < length(@list1); $i++) {
        push(@result, $list1[$i] + $list2[$i]);
    }
    return(\@result);
}
```

```
# Python version.
def pairwise_sum(list1, list2):
    result = []
    for i in range(len(list1)):
        result.append(list1[i] + list2[i])
    return result
```

Both pieces of code do the same thing, but the Python code wins in terms of readability.

1.2.4 Python is complete—"batteries included"

Another advantage of Python is its "batteries included" philosophy when it comes to libraries. The idea is that when you install Python, you should have everything you need to do real work, without the need to install additional libraries. This is why the Python standard library comes with modules for handling email, web pages, databases, operating system calls, GUI development, and more.

For example, with Python, you can write a web server to share the files in a directory with just two lines of code:

```
import http.server
http.server.test(HandlerClass=http.server.SimpleHTTPRequestHandler)
```

There's no need to install libraries to handle network connections and HTTP—it's already in Python, right out of the box.

1.2.5 Python is cross-platform

Python is also an excellent cross-platform language. Python runs on many different platforms: Windows, Mac, Linux, UNIX, and so on. Because it's interpreted, the same code can run on any platform that has a Python interpreter, and almost all current platforms have one. There are even versions of Python that run on Java (Jython) and .NET (IronPython), giving you even more possible platforms that run Python.

1.2.6 Python is free

Python is also free. Python was originally, and continues to be, developed under the open source model, and it's freely available. You can download and install practically any version of Python and use it to develop software for commercial or personal applications, and you don't need to pay a dime.

Although attitudes are changing, some people are still leery of free software because of concerns about a lack of support, fearing they lack the clout of a paying customer. But Python is used by many established companies as a key part of their business; Google, Rackspace, Industrial Light & Magic, and Honeywell are just a few examples. These companies and many others know Python for what it is—a very stable, reliable, and well-supported product with an active and knowledgeable user community. You'll get an answer to even the most difficult Python question more quickly on the Python internet newsgroup than you will on most tech-support phone lines, and the Python answer will be free and correct.

> **Python and open source software**
>
> Not only is Python free of cost, but its source code is also freely available, and you're free to modify, improve, and extend it if you want. Because the source code is freely available, you have the ability to go in yourself and change it (or to hire someone to go in and do so for you). You rarely have this option at any reasonable cost with proprietary software.
>
> If this is your first foray into the world of open source software, you should understand that not only are you free to use and modify Python, but you're also able (and encouraged) to contribute to it and improve it. Depending on your circumstances, interests, and skills, those contributions might be financial, as in a donation to the Python Software Foundation (PSF), or they may involve participating in one of the special interest groups (SIGs), testing and giving feedback on releases of the Python core or one of the auxiliary modules, or contributing some of what you or your company develops back to the community. The level of contribution (if any) is, of course, up to you; but if you're able to give back, definitely consider doing so. Something of significant value is being created here, and you have an opportunity to add to it.

Python has a lot going for it: expressiveness, readability, rich included libraries, and cross-platform capabilities, plus it's open source. What's the catch?

1.3 What Python doesn't do as well

Although Python has many advantages, no language can do everything, so Python isn't the perfect solution for all your needs. To decide whether Python is the right language for your situation, you also need to consider the areas where Python doesn't do as well.

1.3.1 Python is not the fastest language

A possible drawback with Python is its speed of execution. It isn't a fully compiled language. Instead, it's first semicompiled to an internal byte-code form, which is then executed by a Python interpreter. There are some tasks, such as string parsing using regular expressions, for which Python has efficient implementations and is as fast as, or faster than, any C program you're likely to write. Nevertheless, most of the time, using Python results in slower programs than a language like C. But you should keep this in perspective. Modern computers have so much computing power that for the vast majority of applications, the speed of the program isn't as important as the speed of development, and Python programs can typically be written much more quickly. In addition, it's easy to extend Python with modules written in C or C++, which can be used to run the CPU-intensive portions of a program.

1.3.2 Python doesn't have the most libraries

Although Python comes with an excellent collection of libraries, and many more are available, Python doesn't hold the lead in this department. Languages like C, Java, and Perl have even larger collections of libraries available, in some cases offering a solution where Python has none or a choice of several options where Python might have only one. These situations tend to be fairly specialized, however, and Python is easy to extend, either in Python itself or by using existing libraries in C and other languages. For almost all common computing problems, Python's library support is excellent.

1.3.3 Python doesn't check variable types at compile time

Unlike some languages, Python's variables are more like labels that reference various objects: integers, strings, class instances, whatever. That means that although those objects themselves have types, the variables referring to them aren't bound to that particular type. It's possible (if not necessarily desirable) to use the variable `x` to refer to a string in one line and an integer in another:

```
>>> x = "2"
>>> print(x)
'2'                         <-- x is string "2"
>>> x = int(x)
>>> print(x)
2                           <-- x is now integer 2
```

The fact that Python associates types with objects and not with variables means that the interpreter doesn't help you catch variable type mismatches. If you intend a variable `count` to hold an integer, Python won't complain if you assign the string "two" to it. Traditional coders count this as a disadvantage, because you lose an additional free check on your code. But errors like this usually aren't hard to find and fix, and Python's testing features makes avoiding type errors manageable. Most Python programmers feel that the flexibility of dynamic typing more than outweighs the cost.

1.4 Why learn Python 3?

Python has been around for a number of years and has evolved over that time. The first edition of this book was based on Python 1.5.2, and Python 2.x has been the dominant version for several years. This book is based on Python 3.1.

Python 3, originally whimsically dubbed Python 3000, is notable because it's the first version of Python in the history of the language to break backward compatibility. What this means is that code written for earlier versions of Python probably won't run on Python 3 without some changes. In earlier versions of Python, for example, the `print` statement didn't require parentheses around its arguments:

```
print "hello"
```

In Python 3, `print` is a function and needs the parentheses:

```
print("hello")
```

You may be thinking, "Why change details like this, if it's going to break old code?" Because this kind of change is a big step for any language, the core developers of Python thought about this issue carefully. Although the changes in Python 3 break compatibility with older code, those changes are fairly small and for the better—they make the language more consistent, more readable, and less ambiguous. Python 3 isn't a dramatic rewrite of the language; it's a well-thought-out evolution. The core developers also took care to provide a strategy and tools to safely and efficiently migrate old code to Python 3, which will be discussed in a later chapter.

Why learn Python 3? Because it's the best Python so far; and as projects switch to take advantage of its improvements, it will be the dominant Python version for years to come. If you need a library that hasn't been converted yet, by all means stick with Python 2.x; but if you're starting to learn Python or starting a project, then go with Python 3—not only is it better, but it's the future.

1.5 Summary

Python is a modern, high-level language, with many features:

- Dynamic typing
- Simple, consistent syntax and semantics
- Multiplatform
- Well-planned design and evolution of features
- Highly modular
- Suited for both rapid development and large-scale programming
- Reasonably fast and easily extended with C or C++ modules for higher speeds
- Easy access to various GUI toolkits
- Built-in advanced features such as persistent object storage, advanced hash tables, expandable class syntax, universal comparison functions, and so forth
- Powerful included libraries such as numeric processing, image manipulation, user interfaces, web scripting, and others
- Supported by a dynamic Python community
- Can be integrated with a number of other languages to let you take advantage of the strengths of both while obviating their weaknesses

Let's get going. The first step is to make sure you have Python 3 installed on your machine. In the next chapter, we'll look at how to get Python up and running on Windows, Mac, and Linux platforms.

2 Getting started

This chapter covers

- Installing Python
- Using IDLE and the basic interactive mode
- Writing a simple program
- Using IDLE's Python shell window

This chapter guides you through downloading, installing, and starting up Python and IDLE, an integrated development environment for Python. At the time of this writing, the Python language is fairly mature, and version 3.1 has just been released. After going through years of refinement, Python 3 is the first version of the language that isn't fully backward compatible with earlier versions. It should be several years before another such dramatic change occurs, and any future enhancements will be developed with concern to avoid impacting an already significant existing code base. Therefore, the material presented after this chapter isn't likely to become dated anytime soon.

2.1 Installing Python

Installing Python is a simple matter, regardless of which platform you're using. The first step is to obtain a recent distribution for your machine; the most recent one can always be found at www.python.org. This book is based on Python 3.1.

Having more than one version of Python

You may already have an earlier version of Python installed on your machine. Many Linux distributions and Mac OS X come with Python 2.x as part of the operating system. Because Python 3 isn't completely compatible with Python 2, it's reasonable to wonder if installing both versions on the same computer will cause a conflict.

There's no need to worry; you can have multiple versions of Python on the same computer. In the case of UNIX-based systems like OS X and Linux, Python 3 installs alongside the older version and doesn't replace it. When your system looks for "python," it still finds the one it expects; and when you want to access Python 3, you can run `python3.1` or `idle3.1`. In Windows, the different versions are installed in separate locations and have separate menu entries.

Some basic platform-specific descriptions for the Python installation are given next. The specifics can vary quite a bit depending on your platform, so be sure to read the instructions on the download pages and with the various versions. You're probably familiar with installing software on your particular machine, so I'll keep these descriptions short:

- *Microsoft Windows*—Python can be installed in most versions of Windows by using the Python installer program, currently called python-3.1.msi. Just download it, execute it, and follow the installer's prompts. You may need to be logged in as administrator to run the install. If you're on a network and don't have the administrator password, ask your system administrator to do the installation for you.
- *Macintosh*—You need to get a version of Python 3 that matches your OS X version and your processor. After you determine the correct version, download the disk image file, double-click to mount it, and run the installer inside. The OS X installer sets up everything automatically, and Python 3 will be in a subfolder inside the Applications folder, labeled with the version number. Mac OS X ships with various versions of Python as part of the system, but you don't need to worry about that—Python 3 will be installed *in addition to* the system version. You can find more information about using Python on OS X by following the links on the Python home page.
- *Linux/UNIX*—Most Linux distributions come with Python installed. But the versions of Python vary, and the version of Python installed may not be version 3; for this book, you need to be sure you have the Python 3 packages installed. It's also possible that IDLE isn't installed by default, and you'll need to install that package separately. Although it's also possible to build Python 3 from the source code available on the www.python.org website, a number of additional libraries are needed, and the process isn't for novices. If a precompiled version of Python exists for your distribution of Linux, I recommend using that. Use

the software management system for your distribution to locate and install the correct packages for Python 3 and IDLE. Versions are also available for running Python under many other operating systems. See www.python.org for a current list of supported platforms and specifics on installation.

2.2 IDLE and the basic interactive mode

You have two built-in options for obtaining interactive access to the Python interpreter: the original basic (command-line) mode and IDLE. IDLE is available on many platforms, including Windows, Mac, and Linux, but it may not be available on others. You may need to do more work and install additional software packages to get IDLE running, but it will be worth it because it's a large step up from the basic interactive mode. On the other hand, even if you normally use IDLE, at times you'll likely want to fire up the basic mode. You should be familiar enough to start and use either one.

2.2.1 The basic interactive mode

The basic interactive mode is a rather primitive environment. But the interactive examples in this book are generally small; and later in this book, you'll learn how to easily bring code you've placed in a file into your session (using the module mechanism). Let's look at how to start a basic session on Windows, Mac OS X, and UNIX:

- *Starting a basic session on Windows*—For version 3.x of Python, you navigate to the Python (command-line) entry on the Python 3.x submenu of the Programs folder on the Start menu, and click it. Alternatively, you can directly find the Python.exe executable (for example, in C:\Python31) and double-click it. Doing so brings up the window shown in figure 2.1.
- *Starting a basic session on Mac OS X*—Open a terminal window, and type `python3`. If you get a "Command not found" error, run the `Update Shell Profile.command` script found in the Python3 subfolder in the Applications folder.
- *Starting a basic session on UNIX*—Type `python3.1` at a command prompt. A version message similar to the one shown in figure 2.1 followed by the Python prompt >>> appears in the current window.

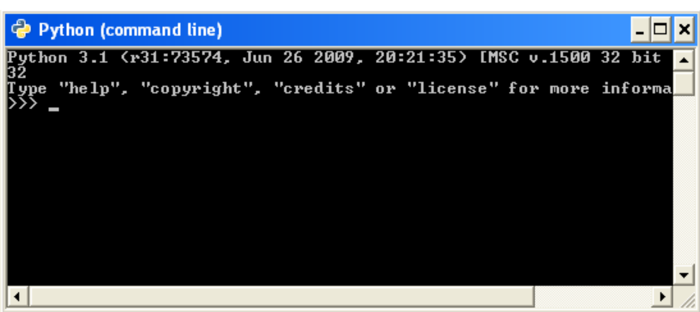


Figure 2.1 Basic interactive mode on Windows XP

Exiting the interactive shell

To exit from a basic session, press Ctrl-Z (if you're on Windows) or Ctrl-D (if you're on Linux or UNIX) or type `exit()` at a command prompt.

Most platforms have a command-line-editing and command-history mechanism. You can use the up and down arrows, as well as the Home, End, Page Up, and Page Down keys, to scroll through past entries and repeat them by pressing the Enter key. (See "Basic Python interactive mode summary" at the end of the appendix.) This is all you need to work your way through this book as you're learning Python. Another option is to use the excellent Python mode available for Emacs, which, among other things, provides access to the interactive mode of Python through an integrated shell buffer.

2.2.2 *The IDLE integrated development environment*

IDLE is the built-in development environment for Python. Its name is based on the acronym for *integrated development environment* (of course, it may have been influenced by the last name of a certain cast member of a particular British television show). IDLE combines an interactive interpreter with code editing and debugging tools to give you one-stop shopping as far as creating Python code is concerned. IDLE's various tools make it an attractive place to start as you learn Python. Let's look at how you run IDLE on Windows, Mac OS X, and Linux:

- *Starting IDLE on Windows*—For version 3.1 of Python, you navigate to the IDLE (Python GUI) entry of the Python 3.1 submenu of the Programs folder of your Start menu, and click it. Doing so brings up the window shown in figure 2.2.
- *Starting IDLE on Mac OS X*—Navigate to the Python 3.x subfolder in the Applications folder, and run IDLE from there.
- *Starting IDLE on Linux or UNIX*—Type `idle3.1` at a command prompt. This brings up a window similar to the one shown in figure 2.2. If you installed IDLE through your distribution's package manager, there should also be a menu entry for IDLE under the Programming submenu or something similar.

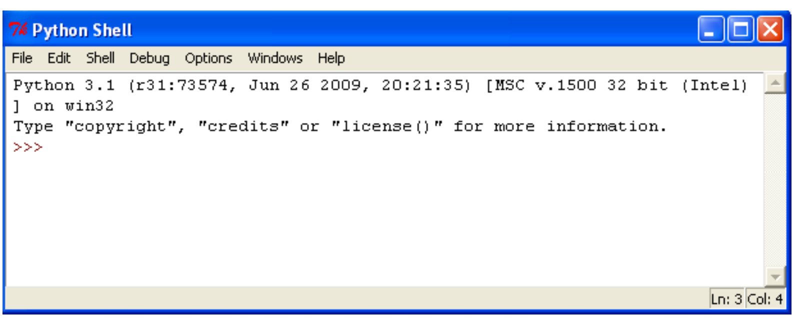


Figure 2.2 IDLE on Windows

2.2.3 Choosing between basic interactive mode and IDLE

Which should you use, IDLE or the basic shell window? To begin, use either IDLE or the Python Shell window. Both have all you need to work through the code examples in this book until you reach chapter 10. From there, we'll cover writing your own modules, and IDLE will be a convenient way to create and edit files. But if you have a strong preference for another editor, you may find that a basic shell window and your favorite editor serve you just as well. If you don't have any strong editor preferences, I suggest using IDLE from the beginning.

2.3 Using IDLE's Python Shell window

The Python Shell window (figure 2.3) opens when you fire up IDLE. It provides automatic indentation and colors your code as you type it in, based on Python syntax types.

You can move around the buffer using the mouse, the arrow keys, the Page Up and Page Down keys, and/or a number of the standard Emacs key bindings. Check the Help menu for the details.

Everything in your session is buffered. You can scroll or search up, place the cursor on any line, and press Enter (creating a hard return), and that line will be copied to the bottom of the screen, where you can edit it and then send it to the interpreter by pressing the Enter key again. Or, leaving the cursor at the bottom, you can toggle up and down through the previously entered commands using Alt-P and Alt-N. This will successively bring copies of the lines to the bottom. When you have the one you want, you can again edit it and then send it to the interpreter by pressing the Enter key. You can complete Python keywords or user-defined values by pressing Alt-/.

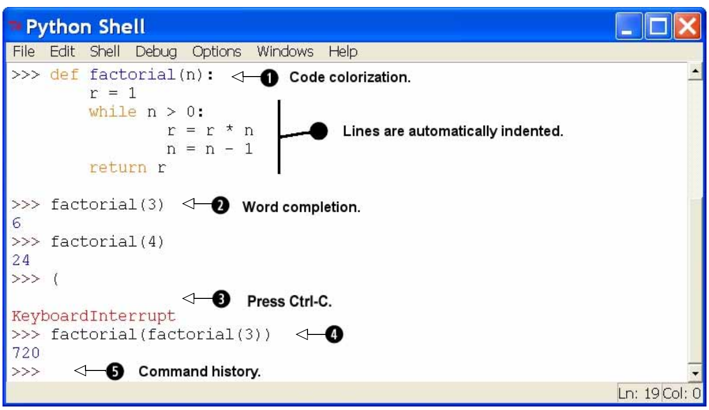


Figure 2.3 Using the Python shell in IDLE. ❶ Code is automatically colored (based on Python syntax) as it's typed in. ❷ Here I typed `f` and then pressed Alt-/, and automatic completion finished the word *factorial*. ❸ I lost the prompt, so I pressed Ctrl-C to interrupt the interpreter and get the prompt back (a closed bracket would have worked here as well). ❹ Placing the cursor on any previous command and pressing the Enter key moves the command and the cursor to the bottom, where you can edit the command and then press Enter to send it to the interpreter. ❺ Placing the cursor at the bottom, you can toggle up and down through the history of previous commands using Alt-P and Alt-N. When you have the command you want, edit it as desired and press Enter, and it will be sent to the interpreter.

If you ever find yourself in a situation where you seem to be hung and can't get a new prompt, the interpreter is likely in a state where it's waiting for you to enter something specific. Pressing Ctrl-C sends an interrupt and should get you back to a prompt. It can also be used to interrupt any running command. To exit IDLE, choose Exit from the File menu.

The Edit menu is the one you'll likely be using the most to begin with. Like any of the other menus, you can tear it off by double-clicking the dotted line at its top and leaving it up beside your window.

2.4 Hello, world

Regardless of how you're accessing Python's interactive mode, you should see a prompt consisting of three angle braces: `>>>`. This is the Python command prompt, and it indicates that you can type in a command to be executed or an expression to be evaluated. Start with the obligatory "Hello, World" program, which is a one-liner in Python. (End each line you type with a hard return.)

```
>>> print("Hello, World")
Hello, World
```

Here I entered the `print` command at the command prompt, and the result appeared on the screen.

Executing the `print` function causes its argument to be printed to the standard output, usually the screen. If the command had been executed while Python was running a Python program from a file, exactly the same thing would have happened: "Hello, World" would have been printed to the screen.

Congratulations! You've just written your first Python program, and we haven't even started talking about the language.

2.5 Using the interactive prompt to explore Python

Whether you're in IDLE or at a standard interactive prompt, there are a couple of handy tools to help you explore Python. The first is the `help()` function, which has two modes. You can just enter `help()` at the prompt to enter the help system, where you can get help on modules, keywords, or topics. When you're in the help system, you see a `help>` prompt, and you can enter a module name, such as `math` or some other topic, to browse Python's documentation on that topic.

Usually it's more convenient to use `help()` in a more targeted way. Entering a type or variable name as a parameter for `help()` gives you an immediate display of that type's documentation:

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
```

```
 |  Convert a string or number to an integer, if possible.  A floating
 |  point argument will be truncated towards zero (this does not include a
 |  string representation of a floating point number!)  When converting a
 |  string, use the optional base.  It is an error to supply a base when
 |  converting a non-string.
 |
 |  Methods defined here:
... (continues with a list of methods for an int)
```

Using `help()` in this way is handy for checking the exact syntax of a method or the behavior of an object.

The `help()` function is part of the `pydoc` library, which has several options for accessing the documentation built into Python libraries. Because every Python installation comes with complete documentation, you can have all of the official documentation at your fingertips, even if you aren't online. See the appendix for more information on accessing Python's documentation.

The other useful function is `dir()`, which lists the objects in a particular namespace. Used with no parameters, it lists the current globals, but it can also list objects for a module or even a type:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'x']
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
     '__delattr__', '__divmod__', '__doc__', '__eq__', '__float__',
     '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
     '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
     '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
     '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
     '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
     '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
     '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
     '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
     '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
     '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator',
     'imag', 'numerator', 'real']
>>>
```

`dir()` is useful for finding out what methods and data are defined, for reminding yourself at a glance of all the members that belong an object or module, and for debugging, because you can see what is defined where.

You can use two other functions to see local and global variables, respectively. They are `globals` and `locals`:

```
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
     '__doc__': None, '__package__': None}
```

Unlike `dir`, both `globals` and `locals` show the values associated with the objects. You'll find out more about both of these functions in chapter 10; for now, it's enough

that you're aware that you have several options for examining what's going on within a Python session.

2.6 Summary

Installing Python 3 is the first step. On Windows systemsm it's as simple as downloading the latest installer from python.org and running it. On Linux, UNIX, and Mac systems the steps will vary. Check for instructions on the www.python.org website, and use your system's software package installer where possible.

After you've installed Python, you can use either the basic interactive shell (and later, your favorite editor) or the IDLE integrated development environment. Whichever you decide to use, it's time to move to chapter 3, where we'll make a quick survey of Python the language.

3 The Quick Python overview

This chapter covers

- Surveying Python
- Using built-in data types
- Controlling program flow
- Creating modules
- Using object-oriented programming

The purpose of this chapter is to give you a basic feeling for the syntax, semantics, capabilities, and philosophy of the Python language. It has been designed to provide you with an initial perspective or conceptual framework on which you'll be able to add details as you encounter them in the rest of the book.

On an initial read, you needn't be concerned about working through and understanding the details of the code segments. You'll be doing fine if you pick up a bit of an idea about what is being done. The subsequent chapters of this book will walk you through the specifics of these features and won't assume prior knowledge. You can always return to this chapter and work through the examples in the appropriate sections as a review after you've read the later chapters.

3.1 Python synopsis

Python has a number of built-in data types such as integers, floats, complex numbers, strings, lists, tuples, dictionaries, and file objects. These can be manipulated using language operators, built-in functions, library functions, or a data type's own methods.

Programmers can also define their own classes and instantiate their own class instances.[1] These can be manipulated by programmer-defined methods as well as the language operators and built-in functions for which the programmer has defined the appropriate special method attributes.

Python provides conditional and iterative control flow through an `if-elif-else` construct along with `while` and `for` loops. It allows function definition with flexible argument-passing options. Exceptions (errors) can be raised using the `raise` statement and caught and handled using the `try-except-else` construct.

Variables don't have to be declared and can have any built-in data type, user-defined object, function, or module assigned to them.

3.2 Built-in data types

Python has several built-in data types, from scalars like numbers and Booleans, to more complex structures like lists, dictionaries, and files.

3.2.1 Numbers

Python's four number types are integers, floats, complex numbers, and Booleans:

- *Integers*—1, –3, 42, 355, 888888888888888, –7777777777
- *Floats*—3.0, 31e12, –6e-4
- *Complex numbers*—3 + 2j, –4- 2j, 4.2 + 6.3j
- *Booleans*—True, False

You can manipulate them using the arithmetic operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `**` (exponentiation), and `%` (modulus).

The following examples use integers:

```
>>> x = 5 + 2 - 3 * 2
>>> x
1
>>> 5 / 2
2.5                                    ❶
>>> 5 // 2
2                                      ❷
>>> 5 % 2
1
>>> 2 ** 8
256
>>> 1000000001 ** 3
1000000003000000003000000001           ❸
```

[1] The Python documentation and this book use the term *object* to refer to instances of any Python data type, not just what many other languages would call *class instances*. This is because all Python objects are instances of one class or another.

Division of integers with / ❶ results in a float (new in Python 3.x), and division of integers with // ❷ results in truncation. Note that integers are of unlimited size ❸; they will grow as large as you need them to.

These examples work with floats, which are based on the doubles in C:

```
>>> x = 4.3 ** 2.4
>>> x
33.137847377716483
>>> 3.5e30 * 2.77e45
9.6950000000000002e+75
>>> 1000000001.0 ** 3
1.000000003e+27
```

Next, the following examples use complex numbers:

```
>>> (3+2j) ** (2+3j)
(0.68176651908903363-2.1207457766159625j)
>>> x = (3+2j) * (4+9j)        ❶
>>> x
(-6+35j)
>>> x.real
-6.0
>>> x.imag
35.0
```

Complex numbers consist of both a real element and an imaginary element, suffixed with a `j`. In the preceding code, variable `x` is assigned to a complex number ❶. You can obtain its "real" part using the attribute notation `x.real`.

Several built-in functions can operate on numbers. There are also the library module `cmath` (which contains functions for complex numbers) and the library module `math` (which contains functions for the other three types):

```
>>> round(3.49)          ❶ Built-in function
3
>>> import math
>>> math.ceil(3.49)      ❷ Library module function
4
```

Built-in functions are always available and are called using a standard function calling syntax. In the preceding code, `round` is called with a float as its input argument ❶.

The functions in library modules are made available using the `import` statement. At ❷, the `math` library module is imported, and its `ceil` function is called using attribute notation: *`module.function(arguments)`*.

The following examples use Booleans:

```
>>> x = False
>>> x
False
>>> not x
True
>>> y = True * 2        ❶
>>> y
2
```

Other than their representation as `True` and `False`, Booleans behave like the numbers 1 (True) and 0 (False) ❶.

3.2.2 Lists

Python has a powerful built-in list type:

```
[]
[1]
[1, 2, 3, 4, 5, 6, 7, 8, 12]
[1, "two", 3L, 4.0, ["a", "b"], (5,6)]      ❶
```

A list can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number ❶.

A list can be indexed from its front or back. You can also refer to a subsegment, or *slice*, of a list using slice notation:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'                     ❶
>>> x[2]
'third'
>>> x[-1]
'fourth'
>>> x[-2]                    ❷
'third'
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']      ❸
>>> x[-2:-1]
['third']
>>> x[:3]
['first', 'second', 'third']      ❹
>>> x[-2:]
['third', 'fourth']
```

Index from the front ❶ using positive indices (starting with 0 as the first element). Index from the back ❷ using negative indices (starting with -1 as the last element). Obtain a slice using `[m:n]` ❸, where `m` is the inclusive starting point and `n` is the exclusive ending point (see table 3.1). An `[:n]` slice ❹ starts at its beginning, and an `[m:]` slice goes to a list's end.

Table 3.1 List indices

x=	[	"first" ,	"second" ,	"third" ,	"fourth"	]
Positive indices		0	1	2	3	
Negative indices		–4	–3	–2	–1	

You can use this notation to add, remove, and replace elements in a list or to obtain an element or a new list that is a slice from it:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1] = "two"
>>> x[8:9] = []
>>> x
[1, 'two', 3, 4, 5, 6, 7, 8]
>>> x[5:7] = [6.0, 6.5, 7.0]            ❶
>>> x
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
>>> x[5:]
[6.0, 6.5, 7.0, 8]
```

The size of the list increases or decreases if the new slice is bigger or smaller than the slice it's replacing ❶.

Some built-in functions (`len`, `max`, and `min`), some operators (`in`, `+`, and `*`), the `del` statement, and the list methods (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, and `sort`) will operate on lists:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(x)
9
>>> [-1, 0] + x                          ❶
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.reverse()                          ❷
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

The operators `+` and `*` each create a new list, leaving the original unchanged ❶. A list's methods are called using attribute notation on the list itself: `x.`*`method`*`(`*`arguments`*`)` ❷.

A number of these operations repeat functionality that can be performed with slice notation, but they improve code readability.

3.2.3 Tuples

Tuples are similar to lists but are *immutable*—that is, they can't be modified after they have been created. The operators (`in`, `+`, and `*`) and built-in functions (`len`, `max`, and `min`), operate on them the same way as they do on lists, because none of them modify the original. Index and slice notation work the same way for obtaining elements or slices but can't be used to add, remove, or replace elements. There are also only two tuple methods: `count` and `index`. A major purpose of tuples is for use as keys for dictionaries. They're also more efficient to use when you don't need modifiability.

```
()
(1,)                                     ❶
(1, 2, 3, 4, 5, 6, 7, 8, 12)
(1, "two", 3L, 4.0, ["a", "b"], (5, 6))  ❷
```

A one-element tuple ❶ needs a comma. A tuple, like a list, can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number ❷.

A list can be converted to a tuple using the built-in function `tuple`:

```
>>> x = [1, 2, 3, 4]
>>> tuple(x)
(1, 2, 3, 4)
```

Conversely, a tuple can be converted to a list using the built-in function `list`:

```
>>> x = (1, 2, 3, 4)
>>> list(x)
[1, 2, 3, 4]
```

3.2.4 Strings

String processing is one of Python's strengths. There are many options for delimiting strings:

```
"A string in double quotes can contain 'single quote' characters."
'A string in single quotes can contain "double quote" characters.'
'''\This string starts with a tab and ends with a newline character.\n'''
"""This is a triple double quoted string, the only kind that can
    contain real newlines."""
```

Strings can be delimited by single (`' '`), double (`" "`), triple single (`''' '''`), or triple double (`""" """`) quotations and can contain tab (`\t`) and newline (`\n`) characters.

Strings are also immutable. The operators and functions that work with them return new strings derived from the original. The operators (`in`, `+`, and `*`) and built-in functions (`len`, `max`, and `min`) operate on strings as they do on lists and tuples. Index and slice notation works the same for obtaining elements or slices but can't be used to add, remove, or replace elements.

Strings have several methods to work with their contents, and the `re` library module also contains functions for working with strings:

```
>>> x = "live and     let \t   \tlive"
>>> x.split()
['live', 'and', 'let', 'live']
>>> x.replace("    let \t   \tlive", "enjoy life")
'live and enjoy life'
>>> import re                                          ❶
>>> regexpr = re.compile(r"[\t ]+")
>>> regexpr.sub(" ", x)
'live and let live'
```

The `re` module ❶ provides regular expression functionality. It provides more sophisticated pattern extraction and replacement capability than the `string` module.

The `print` function outputs strings. Other Python data types can be easily converted to strings and formatted:

```
>>> e = 2.718
>>> x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
>>> print("The constant e is:", e, "and the list x is:", x)      ❶
```

```
The constant e is: 2.718 and the list x is: [1, 'two', 3, 4.0,
['a', 'b'], (5, 6)]
>>> print("the value of %s is: %.2f" % ("e", e))          ❷
the value of e is: 2.72
```

Objects are automatically converted to string representations for printing ❶. The `%` operator ❷ provides a formatting capability similar to that of C's `sprintf`.

3.2.5 Dictionaries

Python's built-in dictionary data type provides associative array functionality implemented using hash tables. The built-in `len` function returns the number of key-value pairs in a dictionary. The `del` statement can be used to delete a key-value pair. As is the case for lists, a number of dictionary methods (`clear`, `copy`, `get`, `has_key`, `items`, `keys`, `update`, and `values`) are available.

```
>>> x = {1: "one", 2: "two"}
>>> x["first"] = "one"
>>> x[("Delorme", "Ryan", 1995)] = (1, 2, 3)          ❶
>>> list(x.keys())
['first', 2, 1, ('Delorme', 'Ryan', 1995)]
>>> x[1]
'one'
>>> x.get(1, "not available")
'one'
>>> x.get(4, "not available")          ❷
'not available'
```

Keys must be of an immutable type ❶. This includes numbers, strings, and tuples. Values can be any kind of object, including mutable types such as lists and dictionaries. The dictionary method `get` ❷ optionally returns a user-definable value when a key isn't in a dictionary.

3.2.6 Sets

A set in Python is an unordered collection of objects, used in situations where membership and uniqueness in the set are the main things you need to know about that object. You can think of sets as a collection of dictionary keys without any associated values:

```
>>> x = set([1, 2, 3, 1, 3, 5])          ❶
>>> x
{1, 2, 3, 5}          ❷
>>> 1 in x          ❸
True
>>> 4 in x          ❸
False
>>>
```

You can create a set by using `set` on a sequence, like a list ❶. When a sequence is made into a set, duplicates are removed ❷. The `in` keyword ❸ is used to check for membership of an object in a set.

3.2.7 File objects

A file is accessed through a Python file object:

```
>>> f = open("myfile", "w")                                          ❶
>>> f.write("First line with necessary newline character\n")
44
>>> f.write("Second line to write to the file\n")
33
>>> f.close()
>>> f = open("myfile", "r")                                          ❷
>>> line1 = f.readline()
>>> line2 = f.readline()
>>> f.close()
>>> print(line1, line2)
First line with necessary newline character
Second line to write to the file
>>> import os                                                        ❸
>>> print(os.getcwd())
c:\My Documents\test
>>> os.chdir(os.path.join("c:", "My Documents", "images"))          ❹
>>> filename = os.path.join("c:", "My Documents",                    ❺
"test", "myfile")
>>> print(filename)
c:\My Documents\test\myfile
>>> f = open(filename, "r")
>>> print(f.readline())
First line with necessary newline character
>>> f.close()
```

The `open` statement ❶ creates a file object. Here the file myfile in the current working directory is being opened in write (`"w"`) mode. After writing two lines to it and closing it ❷, we open the same file again, this time in the read (`"r"`) mode. The `os` module ❸ provides a number of functions for moving around the file system and working with the pathnames of files and directories. Here, we move to another directory ❹. But by referring to the file by an absolute pathname ❺, we are still able to access it.

A number of other input/output capabilities are available. You can use the built-in `input` function to prompt and obtain a string from the user. The `sys` library module allows access to `stdin`, `stdout`, and `stderr`. The `struct` library module provides support for reading and writing files that were generated by or are to be used by C programs. The `Pickle` library module delivers data persistence through the ability to easily read and write the Python data types to and from files.

3.3 Control flow structures

Python has a full range of structures to control code execution and program flow, including common branching and looping structures.

3.3.1 Boolean values and expressions

Python has several ways of expressing Boolean values; the Boolean constant `False`, `0`, the Python nil value `None`, and empty values (for example, the empty list `[ ]` or empty

string `""`) are all taken as `False`. The Boolean constant `True` and everything else are considered `True`.

You can create comparison expressions using the comparison operators (`<`, `<=`, `==`, `>`, `>=`, `!=`, `is`, `is not`, `in`, `not in`) and the logical operators (`and`, `not`, `or`), which all return `True` or `False`.

3.3.2 The if-elif-else statement

The block of code after the first `true` condition (of an `if` or an `elif`) is executed. If none of the conditions is `true`, the block of code after the `else` is executed:

```
x = 5
if x < 5:
    y = -1
    z = 5
elif x > 5:            ❶
    y = 1
    z = 11
else:
    y = 0              ❷
    z = 10
print(x, y, z)
```

The `elif` and `else` clauses are optional ❶, and there can be any number of `elif` clauses. Python uses indentation to delimit blocks ❷. No explicit delimiters such as brackets or braces are necessary. Each block consists of one or more statements separated by newlines. These statements must all be at the same level of indentation.

The output here would be `5 0 10`.

3.3.3 The while loop

The `while` loop is executed as long as the condition (which here is `x > y`) is `true`:

```
u, v, x, y = 0, 0, 100, 30      ❶
while x > y:
    u = u + y
    x = x - y
    if x < y + 2:
        v = v + x               ❷
        x = 0
    else:
        v = v + y + 2
        x = x - y - 2
print(u, v)
```

This is a shorthand notation. Here, `u` and `v` are assigned a value of 0, `x` is set to 100, and `y` obtains a value of 30 ❶. This is the loop block ❷. It's possible for it to contain `break` (which ends the loop) and `continue` statements (which abort the current iteration of the loop).

The output here would be `60 40`.

3.3.4 The for loop

The `for` loop is simple but powerful because it's possible to iterate over any iterable type, such as a list or tuple. Unlike in many languages, Python's `for` loop iterates over each of the items in a sequence, making it more of a `foreach` loop. The following loop finds the first occurrence of an integer that is divisible by 7:

```
item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
for x in item_list:                                    ❶
    if not isinstance(x, int):
        continue                                       ❷
    if not x % 7:
        print("found an integer divisible by seven: %d" % x)
        break                                          ❸
```

`x` is sequentially assigned each value in the list ❶. If `x` isn't an integer, then the rest of this iteration is aborted by the `continue` statement ❷. Flow control continues with `x` set to the next item from the list. After the first appropriate integer is found, the loop is ended by the `break` statement ❸.

The output here would be

```
found an integer divisible by seven: 49
```

3.3.5 Function definition

Python provides flexible mechanisms for passing arguments to functions:

```
>>> def funct1(x, y, z):                          ❶
...     value = x + 2*y + z**2
...     if value > 0:
...         return x + 2*y + z**2                 ❷
...     else:
...         return 0
...
>>> u, v = 3, 4
>>> funct1(u, v, 2)
15
>>> funct1(u, z=v, y=2)                           ❸
23
>>> def funct2(x, y=1, z=1):                      ❹
...     return x + 2 * y + z ** 2
...
>>> funct2(3, z=4)
21
>>> def funct3(x, y=1, z=1, *tup):                ❺
...     print((x, y, z) + tup)
...
>>> funct3(2)
(2, 1, 1)
>>> funct3(1, 2, 3, 4, 5, 6, 7, 8, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> def funct4(x, y=1, z=1, **dictionary):        ❻
...     print(x, y, z, dict)
>>> funct4(1, 2, m=5, n=9, z=3)
1 2 3 {'n': 9, 'm': 5}
```

Functions are defined using the `def` statement ❶. The `return` statement ❷ is what a function uses to return a value. This value can be of any type. If no `return` statement is encountered, Python's `None` value is returned. Function arguments can be entered either by position or by name (keyword). Here `z` and `y` are entered by name ❸. Function parameters can be defined with defaults that are used if a function call leaves them out ❹. A special parameter can be defined that will collect all extra positional arguments in a function call into a tuple ❺. Likewise, a special parameter can be defined that will collect all extra keyword arguments in a function call into a dictionary ❻.

3.3.6 Exceptions

Exceptions (errors) can be caught and handled using the `try-except-finally-else` compound statement. This statement can also catch and handle exceptions you define and raise yourself. Any exception that isn't caught will cause the program to exit. Listing 3.1 shows basic exception handling.

Listing 3.1 File exception.py

```
class EmptyFileError(Exception):                                   ❶
    pass
filenames = ["myfile1", "nonExistent", "emptyFile", "myfile2"]
for file in filenames:
    try:                                      ❷
        f = open(file, 'r')
        line = f.readline()                            ❸
        if line == "":
            f.close()
            raise EmptyFileError("%s: is empty" % file)           ❹
    except IOError as error:
        print("%s: could not be opened: %s" % (file, error.strerror)
    except EmptyFileError as error:
        print(error)                                       ❺
    else:
        print("%s: %s" % (file, f.readline()))
    finally:
        print("Done processing", file)                        ❻
```

Here we define our own exception type inheriting from the base `Exception` type ❶. If an `IOError` or `EmptyFileError` occurs during the execution of the statements in the `try` block, the associated `except` block is executed ❷. This is where an `IOError` might be raised ❸. Here we raise the `EmptyFileError` ❹. The `else` clause is optional ❺. It's executed if no exception occurs in the `try` block (note that in this example, `continue` statements in the `except` blocks could have been used instead). The `finally` clause is optional ❻. It's executed at the end of the block whether an exception was raised or not.

3.4 Module creation

It's easy to create your own modules, which can be imported and used in the same way as Python's built-in library modules. The example in listing 3.2 is a simple module with one function that prompts the user to enter a filename and determines the number of times words occur in this file.

Listing 3.2 File wo.py

```
"""wo module. Contains function: words_occur()"""                    ❶
# interface functions                        ❷
def words_occur():
    """words_occur() - count the occurrences of words in a file."""
    # Prompt user for the name of the file to use.
    file_name = input("Enter the name of the file: ")
    # Open the file, read it and store its words in a list.
    f = open(file_name, 'r')                                      ❸
    word_list = f.read().split()
    f.close()
    # Count the number of occurrences of each word in the file.
    occurs_dict = {}
    for word in word_list:
        # increment the occurrences count for this word
        occurs_dict[word] = occurs_dict.get(word, 0) + 1
    # Print out the results.                                     ❹
    print("File %s has %d words (%d are unique)" \
      % (file_name, len(word_list), len(occurs_dict)))
    print(occurs_dict)                                        ❺
if __name__ == '__main__':
    words_occur()
```

Documentation strings are a standard way of documenting modules, functions, methods, and classes ❶. Comments are anything beginning with a # character ❷. `read` returns a string containing all the characters in a file ❸, and `split` returns a list of the words of a string "split out" based on whitespace. You can use a \ to break a long statement across multiple lines ❹. This allows the program to also be run as a script by typing `python wo.py` at a command line ❺.

If you place a file in one of the directories on the module search path, which can be found in `sys.path`, then it can be imported like any of the built-in library modules using the `import` statement:

```
>>> import wo
>>> wo.words_occur()              ❶
```

This function is called ❶ using the same attribute syntax as used for library module functions.

Note that if you change the file wo.py on disk, `import` won't bring your changes in to the same interactive session. You use the `reload` function from the `imp` library in this situation:

```
>>> import imp
>>> imp.reload(wo)
<module 'wo'>
```

For larger projects, there is a generalization of the module concept called *packages*. This allows you to easily group a number of modules together in a directory or directory subtree and import and hierarchically refer to them using a *`package.subpackage.module`* syntax. This entails little more than the creation of a possibly empty initialization file for each package or subpackage.

3.5 Object-oriented programming

Python provides full support for OOP. Listing 3.3 is an example that might be the start of a simple shapes module for a drawing program. It's intended mainly to serve as reference if you're already familiar with object-oriented programming. The callout notes relate Python's syntax and semantics to the standard features found in other languages.

Listing 3.3 File sh.py

```
"""sh module. Contains classes Shape, Square and Circle"""
class Shape:                                              ❶
    """Shape class: has method move"""
    def __init__(self, x, y):                             ❷
        self.x = x                                        ❸
        self.y = y
    def move(self, deltaX, deltaY):                       ❹
        self.x = self.x + deltaX
        self.y = self.y + deltaY
class Square(Shape):
    """Square Class:inherits from Shape"""
    def __init__(self, side=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.side = side
class Circle(Shape):                                      ❺
    """Circle Class: inherits from Shape and has method area"""
    pi = 3.14159                                          ❻
    def __init__(self, r=1, x=0, y=0):
        Shape.__init__(self, x, y)                        ❼
        self.radius = r
    def area(self):
        """Circle area method: returns the area of the circle."""
        return self.radius * self.radius * self.pi
    def __str__(self):                                    ❽
        return "Circle of radius %s at coordinates (%d, %d)"\
               % (self.radius, self.x, self.y)
```

Classes are defined using the `class` keyword ❶. The instance initializer method (constructor) for a class is always called `__init__` ❷. Instance variables `x` and `y` are created and initialized here ❸. Methods, like functions, are defined using the `def` keyword ❹. The first argument of any method is by convention called `self`. When the method is invoked, `self` is set to the instance that invoked the method. Class `Circle` inherits from class `Shape` ❺. This is similar to but not exactly like a standard class variable ❻. A class must, in its initializer, explicitly call the initializer of its base class ❼. The `__str__` method is used by the `print` function ❽. Other special attribute methods permit operator overloading or are employed by built-in methods such as the length (`len`) function.

Importing this file makes these classes available:

```
>>> import sh
>>> c1 = sh.Circle()                          ❶
>>> c2 = sh.Circle(5, 15, 20)
>>> print(c1)
Circle of radius 1 at coordinates (0, 0)
>>> print(c2)                                 ❷
Circle of radius 5 at coordinates (15, 20)
>>> c2.area()
78.539749999999998
>>> c2.move(5,6)                              ❸
>>> print(c2)
Circle of radius 5 at coordinates (20, 26)
```

The initializer is implicitly called, and a circle instance is created ❶. The `print` function implicitly uses the special `__str__` method ❷. Here we see that the `move` method of `Circle`'s parent class `Shape` is available ❸. A method is called using attribute syntax on the object instance: `object.method()`. The first (`self`) parameter is set implicitly.

3.6 Summary

This ends our overview of Python. Don't worry if some parts were confusing. You need an understanding of only the broad strokes at this point. The chapters in part 2 and part 3 won't assume prior knowledge of their concepts and will walk you through these features in detail. You can also think of this as an early preview of what your level of knowledge will be when you're ready to move on to the chapters in part 4. You may find it valuable to return here and work through the appropriate examples as a review after we cover the features in subsequent chapters.

If this chapter was mostly a review for you, or there were only a few features you would like to learn more about, feel free to jump ahead, using the index, the table of contents, or the appendix. You can always slow down if anything catches your eye. You probably should have an understanding of Python to the level that you have no trouble understanding most of this chapter before you move on to the chapters in part 4.

Part 2

The essentials

In the chapters that follow, we'll show you the essentials of Python. We'll start from the absolute basics of what makes a Python program and move through Python's built-in data types and control structures, as well as defining functions and using modules.

The last section of this part moves on to show you how to write standalone Python programs, manipulate files, handle errors, and use classes. The section ends with chapter 16, which is a brief introduction to GUI programming using Python's `tkinter` module.

4 The absolute basics

This chapter covers

- Indenting and block structuring
- Differentiating comments
- Assigning variables
- Evaluating expressions
- Using common data types
- Getting user input
- Using correct Pythonic style

This chapter describes the absolute basics in Python: assignments and expressions, how to type a number or a string, how to indicate comments in code, and so forth. It starts out with a discussion of how Python block structures its code, which is different from any other major language.

4.1 Indentation and block structuring

Python differs from most other programming languages because it uses whitespace and indentation to determine block structure (that is, to determine what constitutes the body of a loop, the `else` clause of a conditional, and so on). Most languages use

braces of some sort to do this. Here is C code that calculates the factorial of 9, leaving the result in the variable `r`:

```
/* This is C code  */
int n, r;
n = 9;
r = 1;
while (n > 0) {
    r *= n;
    n--;
}
```

The { and } delimit the body of the `while` loop, the code that is executed with each repetition of the loop. The code is usually indented more or less as shown, to make clear what's going on, but it could also be written like this:

```
/* And this is C code with arbitrary indentation */
    int n, r;
        n = 9;
        r = 1;
    while (n > 0) {
r *=  n;
n--;
}
```

It still would execute correctly, even though it's rather difficult to read.

Here's the Python equivalent:

```
# This is Python code. (Yea!)
n = 9
r = 1
while n > 0:
    r = r * n
    n = n - 1
```

Python also supports C-style `r *= n`

Python also supports `n -= 1`

Python doesn't use braces to indicate code structure; instead, the indentation itself is used. The last two lines of the previous code are the body of the `while` loop because they come immediately after the `while` statement and are indented one level further than the `while` statement. If they weren't indented, they wouldn't be part of the body of the `while`.

Using indentation to structure code rather than braces may take some getting used to, but there are significant benefits:

- It's impossible to have missing or extra braces. You'll never need to hunt through your code for the brace near the bottom that matches the one a few lines from the top.
- The visual structure of the code reflects its real structure. This makes it easy to grasp the skeleton of code just by looking at it.
- Python coding styles are mostly uniform. In other words, you're unlikely to go crazy from dealing with someone's idea of aesthetically pleasing code. Their code will look pretty much like yours.

You probably use consistent indentation in your code already, so this won't be a big step for you. If you're using IDLE, it automatically indents lines. You just need to backspace out of levels of indentation when desired. Most programming editors and IDEs, including Emacs, VIM, and Eclipse, to name a few, provide this functionality as well. One thing that may trip you up once or twice until you get used to it is that the Python interpreter returns an error message if you have a space (or spaces) preceding the commands you enter at a prompt.

4.2 Differentiating comments

For the most part, anything following a `#` symbol in a Python file is a comment and is disregarded by the language. The obvious exception is a `#` in a string, which is just a character of that string:

```
# Assign 5 to x
x = 5
x = 3                    # Now x is 3
x = "# This is not a comment"
```

We'll put comments into Python code frequently.

4.3 Variables and assignments

The most commonly used command in Python is assignment, which looks pretty close to what you might've used in other languages. Python code to create a variable called x and assign the value 5 to that variable is

```
x = 5
```

In Python, neither a variable type declaration nor an end-of-line delimiter is necessary, unlike in many other computer languages. The line is ended by the end of the line. Variables are created automatically when they're first assigned.

Python variables can be set to any object, unlike C or many other languages' variables, which can store only the type of value they're declared as. The following is perfectly legal Python code:

```
>>> x = "Hello"
>>> print(x)
Hello
>>> x = 5
>>> print(x)
5
```

x starts out referring to the string object `"Hello"` and then refers to the integer object `5`. Of course, this feature can be abused because arbitrarily assigning the same variable name to refer successively to different data types can make code confusing to understand.

A new assignment overrides any previous assignments. The `del` statement deletes the variable. Trying to print the variable's contents after deleting it gives an error the same as if the variable had never been created in the first place.

```
>>> x = 5
>>> print(x)
5
>>> del x
>>> print(x)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Here we have our first look at a *traceback*, which is printed when an error, called an *exception*, has been detected. The last line tells us what exception was detected, which in this case is a `NameError` exception on `x`. After its deletion, `x` is no longer a valid variable name. In this example, the trace returns only `line 1, in <module>` because only the single line has been sent in the interactive mode. In general, the full dynamic call structure of the existing function calls at the time of the occurrence of the error is returned. If you're using IDLE, you obtain the same information with some small differences; it may look something like this:

```
Traceback (most recent call last):

  File "<pyshell#3>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

Chapter 14 will describe this mechanism in more detail. A full list of the possible exceptions and what causes them is in the appendix of this book. Use the index to find any specific exception (such as `NameError`) you receive.

Variable names are case sensitive and can include any alphanumeric character as well as underscores but must start with a letter or underscore. See section 4.10 for more guidance on the Pythonic style for creating variable names.

4.4 Expressions

Python supports arithmetic and similar expressions; these will be familiar to most readers. The following code calculates the average of 3 and 5, leaving the result in the variable `z`:

```
x = 3
y = 5
z = (x + y) / 2
```

Note that unlike the arithmetic rules of C in terms of type coercions, arithmetic operators involving only integers do *not* always return an integer. Even though all the values are integers, division (starting with Python 3) returns a floating-point number, so the fractional part isn't truncated. If you want traditional integer division returning a truncated integer, you can use the `//` instead.

Standard rules of arithmetic precedence apply; if we'd left out the parentheses in the last line, it would've been calculated as `x + (y / 2)`.

Expressions don't have to involve just numerical values; strings, Boolean values, and many other types of objects can be used in expressions in various ways. We'll discuss these in more detail as they're used.

4.5 Strings

You've already seen that Python, like most other programming languages, indicates strings through the use of double quotes. This line leaves the string `"Hello, World"` in the variable `x`:

```
x = "Hello, World"
```

Backslashes can be used to escape characters, to give them special meanings. `\n` means the newline character, `\t` means the tab character, `\\` means a single normal backslash character, and `\"` is a plain double-quote character. It doesn't end the string:

```
x = "\tThis string starts with a \"tab\"."
x = "This string contains a single backslash(\\)."
```

You can use single quotes instead of double quotes. The following two lines do the same thing:

```
x = "Hello, World"
x = 'Hello, World'
```

The only difference is that you don't need to backslash `"` characters in single-quoted strings or `'` characters in double-quoted strings:

```
x = "Don't need a backslash"
x = 'Can\'t get by without a backslash'
x = "Backslash your \" character!"
x = 'You can leave the " alone'
```

You can't split a normal string across lines; this code won't work:

```
# This Python code will cause an ERROR -- you can't split the string
across two lines.
x = "This is a misguided attempt to
put a newline into a string without using backslash-n"
```

But Python offers triple-quoted strings, which let you do this and permit single and double quotes to be included without backslashes:

```
x = """Starting and ending a string with triple " characters
permits embedded newlines, and the use of " and ' without
backslashes"""
```

Now `x` is the entire sentence between the `"""` delimiters. (You can also use triple single quotes—`'''`—instead of triple double quotes to do the same thing.)

Python offers enough string-related functionality that chapter 6 is devoted to the topic.

4.6 Numbers

Because you're probably familiar with standard numeric operations from other languages, this book doesn't contain a separate chapter describing Python's numeric abilities. This section describes the unique features of Python numbers, and the Python documentation lists the available functions.

Python offers four kinds of numbers: *integers*, *floats*, *complex numbers*, and *Booleans*. An integer constant is written as an integer—0, –11, +33, 123456—and has unlimited range, restricted only by the resources of your machine. A float can be written with a decimal point or using scientific notation: 3.14, –2E-8, 2.718281828. The precision of these values is governed by the underlying machine but is typically equal to double (64-bit) types in C. Complex numbers are probably of limited interest and are discussed separately later in the section. Booleans are either `True` or `False` and behave identically to 1 and 0 except for their string representations.

Arithmetic is much like it is in C. Operations involving two integers produce an integer, except for division (`/`), where a float results. If the `//` division symbol is used, the result is an integer, with truncation. Operations involving a float always produce a float. Here are a few examples:

```
>>> 5 + 2 - 3 * 2
1
>>> 5 / 2          # floating point result with normal division
2.5
>>> 5 / 2.0        # also a floating point result
2.5
>>> 5 // 2         # integer result with truncation when divided using '//'
2
>>> 30000000000    # This would be too large to be an int in many languages
30000000000
>>> 30000000000 * 3
90000000000
>>> 30000000000 * 3.0
90000000000.0
>>> 2.0e-8         # Scientific notation gives back a float
2e-08
>>> 3000000 * 3000000
9000000000000
>>> int(200.2)        ❶
200
>>> int(2e2)          ❶
200
>>> float(200)        ❶
200.0
```

These are explicit conversions between types ❶. `int` will truncate float values.

Numbers in Python have two advantages over C or Java. First, integers can be arbitrarily large; and second, the division of two integers results in a float.

4.6.1 Built-in numeric functions

Python provides the following number-related functions as part of its core:

```
abs, divmod, cmp, coerce, float, hex, int, long, max, min, oct,
pow, round
```

See the documentation for details.

4.6.2 Advanced numeric functions

More advanced numeric functions such as the trig and hyperbolic trig functions, as well as a few useful constants, aren't built-ins in Python but are provided in a standard module called `math`. Modules will be explained in detail later; for now, it's sufficient to know that the math functions in this section must be made available by starting your Python program or interactive session with the statement

```
from math import *
```

The `math` module provides the following functions and constants:

```
acos, asin, atan, atan2, ceil, cos, cosh, e, exp, fabs, floor, fmod,
frexp, hypot, ldexp, log, log10, mod, pi, pow, sin, sinh, sqrt, tan,
tanh
```

See the documentation for details.

4.6.3 Numeric computation

The core Python installation isn't well suited to intensive numeric computation because of speed constraints. But the powerful Python extension `NumPy` provides highly efficient implementations of many advanced numeric operations. The emphasis is on array operations, including multidimensional matrices and more advanced functions such as the Fast Fourier Transform. You should be able to find `NumPy` (or links to it) at www.scipy.org.

4.6.4 Complex numbers

Complex numbers are created automatically whenever an expression of the form `nj` is encountered, with `n` having the same form as a Python integer or float. `j` is, of course, standard engineering notation for the imaginary number equal to the square root of –1, for example:

```
>>> (3+2j)
(3+2j)
```

Note that Python expresses the resulting complex number in parentheses, as a way of indicating that what is printed to the screen represents the value of a single object:

```
>>> 3 + 2j - (4+4j)
(-1-2j)
```

```
>>> (1+2j) * (3+4j)
(-5+10j)
>>> 1j * 1j
(-1+0j)
```

Calculating `j * j` gives the expected answer of –1, but the result remains a Python complex number object. Complex numbers are never converted automatically to equivalent real or integer objects. But you can easily access their real and imaginary parts with `real` and `imag`:

```
>>> z = (3+5j)
>>> z.real
3.0
>>> z.imag
5.0
```

Note that real and imaginary parts of a complex number are always returned as floating-point numbers.

4.6.5 Advanced complex-number functions

The functions in the `math` module don't apply to complex numbers; the rationale is that most users want the square root of –1 to generate an error, not an answer! Instead, similar functions, which can operate on complex numbers, are provided in the `cmath` module:

```
acos, acosh, asin, asinh, atan, atanh, cos, cosh, e, exp, log, log10,
pi, sin, sinh, sqrt, tan, tanh.
```

In order to make clear in the code that these are special-purpose complex-number functions and to avoid name conflicts with the more normal equivalents, it's best to import the `cmath` module by saying

```
import cmath
```

and then to explicitly refer to the `cmath` package when using the function:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Minimizing from <module> import *

This is a good example of why it's best to minimize the use of the `from <module> import *` form of the `import` statement. If you imported first the `math` and then the `cmath` modules using it, the commonly named functions in `cmath` would override those of `math`. It's also more work for someone reading your code to figure out the source of the specific functions you use. Some modules are explicitly designed to use this form of import.

See chapter 10 for more details on how to use modules and module names.

The important thing to keep in mind is that by importing the `cmath` module, you can do almost anything you can do with other numbers.

4.7 The None value

In addition to standard types such as strings and numbers, Python has a special basic data type that defines a single special data object called `None`. As the name suggests, `None` is used to represent an empty value. It appears in various guises throughout Python. For example, a procedure in Python is just a function that doesn't explicitly return a value, which means that, by default, it returns `None`.

`None` is often useful in day-to-day Python programming as a placeholder, to indicate a point in a data structure where meaningful data will eventually be found, even though that data hasn't yet been calculated. You can easily test for the presence of `None`, because there is only one instance of `None` in the entire Python system (all references to `None` point to the same object), and `None` is equivalent only to itself.

4.8 Getting input from the user

You can also use the `input()` function to get input from the user. Use the prompt string you want displayed to the user as `input`'s parameter:

```
>>> name = input("Name? ")
Name? Vern
>>> print(name)
Vern
>>> age = int(input("Age? "))     <-- Converts input from string to int
Age? 28
>>> print(age)
28
>>>
```

This is a fairly simple way to get user input. The one catch is that the input comes in as a string, so if you want to use it as a number, you have to use the `int()` or `float()` function to convert it.

4.9 Built-in operators

Python provides various built-in operators, from the standard (such as +, *, and so on) to the more esoteric, such as operators for performing bit shifting, bitwise logical functions, and so forth. Most of these operators are no more unique to Python than to any other language, and hence I won't explain them in the main text. You can find a complete list of the Python built-in operators in the documentation.

4.10 Basic Python style

Python has relatively few limitations on coding style with the obvious exception of the requirement to use indentation to organize code into blocks. Even in that case, the amount of indentation and type of indentation (tabs versus spaces) isn't mandated. However, there are preferred stylistic conventions for Python, which are contained in

Python Enhancement Proposal (PEP) 8, which is summarized in the appendix and can be found online at www.python.org/dev/peps/pep-0008/. A selection of Pythonic conventions is provided in table 4.1, but to fully absorb Pythonic style you'll need to periodically reread PEP 8.

Table 4.1 Pythonic coding conventions

Situation	Suggestion	Example
Module/package names	short, all lowercase, underscores only if needed	`imp, sys`
Function names	all lowercase, underscores_for_readablitiy	`foo(), my_func()`
Variable names	all lowercase, underscores_for_readablitiy	`my_var`
Class names	CapitalizeEachWord	`MyClass`
Constant names	ALL_CAPS_WITH_UNDERSCORES	`PI, TAX_RATE`
Indentation	4 spaces per level, don't use tabs	
Comparisons	Don't compare explicitly to True or False	`if my_var:` `if not my_var:`

I strongly urge you to follow the conventions of PEP 8. They're wisely chosen and time tested and will make your code easier for you and other Python programmers to understand.

4.11 Summary

That's the view of Python from 30,000 feet. If you're an experienced programmer, you're probably already seeing how you can write your code in Python. If that's the case, you should feel free to start experimenting with your own code. Many programmers find it surprisingly easy to pick up Python syntax, because there are relatively few surprises. Once you pick up the basics of the language, it's very predictable and consistent.

In any case, we have just covered the broadest outlines of the language, and there are lots of details that we still need to cover, beginning in the next chapter with one of the workhorses of Python, lists.

5 Lists, tuples, and sets

This chapter covers

- Manipulating lists and list indices
- Modifying lists
- Sorting
- Using common list operations
- Handling nested lists and deep copies
- Using tuples
- Creating and using sets

In this chapter, we'll discuss the two major Python sequence types: lists and tuples. At first, lists may remind you of arrays in many other languages, but don't be fooled—lists are a good deal more flexible and powerful than plain arrays. This chapter also discusses a newer Python collection type: sets. Sets are useful when an object's membership in the collection, as opposed to its position, is important.

Tuples are like lists that can't be modified—you can think of them as a restricted type of list or as a basic record type. We'll discuss why we need such a restricted data type later in the chapter.

Most of the chapter is devoted to lists, because if you understand lists, you pretty much understand tuples. The last part of the chapter discusses the differences between lists and tuples, in both functional and design terms.

5.1 Lists are like arrays

A list in Python is much the same thing as an array in Java or C or any other language. It's an ordered collection of objects. You create a listd by enclosing a comma-separated list of elements in square brackets, like so:

```
# This assigns a three-element list to x
x = [1, 2, 3]
```

Note that you don't have to worry about declaring the list or fixing its size ahead of time. This line creates the list as well as assigns it, and a list automatically grows or shrinks in size as needed.

> **Arrays in Python**
>
> A typed `array` module is available in Python that provides arrays based on C data types. Information on its use can be found in the *Python Library Reference*. I suggest you look into it only if you run into a situation where you really need the performance improvement. If a situation calls for numerical computations, you should consider using `NumPy`, mentioned in section 4.6.3, available at www.scipy.org.

Unlike lists in many other languages, Python lists can contain different types of elements; a list element can be any Python object. Here's a list that contains a variety of elements:

```
# First element is a number, second is a string, third is another list.
x = [2, "two", [1, 2, 3]]
```

Probably the most basic built-in list function is the `len` function, which returns the number of elements in a list:

```
>>> x = [2, "two", [1, 2, 3]]
>>> len(x)
3
```

Note that the `len` function doesn't count the items in the inner, nested list.

5.2 List indices

Understanding how list indices work will make Python much more useful to you. Please read the whole section!

Elements can be extracted from a Python list using a notation like C's array indexing. Like C and many other languages, Python starts counting from 0; asking for element 0

returns the first element of the list, asking for element 1 returns the second element, and so forth. Here are a few examples:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[2]
'third'
```

But Python indexing is more flexible than C indexing; if indices are negative numbers, they indicate positions counting from the end of the list, with –1 being the last position in the list, –2 being the second-to-last position, and so forth. Continuing with the same list x, we can do the following:

```
>>> a = x[-1]
>>> a
'fourth'
>>> x[-2]
'third'
```

For operations involving a single list index, it's generally satisfactory to think of the index as pointing at a particular element in the list. For more advanced operations, it's more correct to think of list indices as indicating positions *between* elements. In the list `["first", "second", "third", "fourth"]`, you can think of the indices as pointing like this:

x =[		**"first",**		**"second",**		**"third",**		**"fourth"**		**]**
Positive indices	0		1		2		3			
Negative indices	–4		–3		–2		–1			

This is irrelevant when you're extracting a single element, but Python can extract or assign to an entire sublist at once, an operation known as *slicing*. Instead of entering `list[index]` to extract the item just after `index`, enter `list[index1:index2]` to extract all items including `index1` and up to (but not including) `index2` into a new list. Here are some examples:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
```

It may seem reasonable that if the second index indicates a position in the list *before* the first index, this would return the elements between those indices in reverse order, but this isn't what happens. Instead, this return: an empty list:

```
>>> x[-1:2]
[]
```

When slicing a list, it's also possible to leave out `index1` or `index2`. Leaving out `index1` means "go from the beginning of the list," and leaving out `index2` means "go to the end of the list":

```
>>> x[:3]
['first', 'second', 'third']
>>> x[2:]
['third', 'fourth']
```

Omitting both indices makes a new list that goes from the beginning to the end of the original list; that is, it copies the list. This is useful when you wish to make a copy that you can modify, without affecting the original list:

```
>>> y = x[:]
>>> y[0] = '1 st'
>>> y
['1 st', 'second', 'third', 'fourth']
>>> x
['first', 'second', 'third', 'fourth']
```

5.3 Modifying lists

You can use list index notation to modify a list as well as to extract an element from it. Put the index on the left side of the assignment operator:

```
>>> x = [1, 2, 3, 4]
>>> x[1] = "two"
>>> x
[1, 'two', 3, 4]
```

Slice notation can be used here too. Saying something like `lista[index1:index2] = listb` causes all elements of `lista` between `index1` and `index2` to be replaced with the elements in `listb`. `listb` can have more or fewer elements than are removed from `lista`, in which case the length of `lista` will be altered. You can use slice assignment to do a number of different things, as shown here:

```
>>> x = [1, 2, 3, 4]
>>> x[len(x):] = [5, 6, 7]        <- Appends list to end of list
>>> x
[1, 2, 3, 4, 5, 6, 7]
>>> x[:0] = [-1, 0]               <- Appends list to front of list
>>> x
[-1, 0, 1, 2, 3, 4, 5, 6, 7]
>>> x[1:-1] = []                  <- Removes elements from list
>>> x
[-1, 7]
```

Appending a single element to a list is such a common operation that there's a special `append` method to do it:

```
>>> x = [1, 2, 3]
>>> x.append("four")
>>> x
[1, 2, 3, 'four']
```

One problem can occur if you try to append one list to another. The list gets appended as a single element of the main list:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.append(y)
>>> x
[1, 2, 3, 4, [5, 6, 7]]
```

The `extend` method is like the `append` method, except that it allows you to add one list to another:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 5, 6, 7]
```

There is also a special `insert` method to insert new list elements between two existing elements or at the front of the list. `insert` is used as a method of lists and takes two additional arguments; the first is the index position in the list where the new element should be inserted, and the second is the new element itself:

```
>>> x = [1, 2, 3]
>>> x.insert(2, "hello")
>>> print(x)
[1, 2, 'hello', 3]
>>> x.insert(0, "start")
>>> print(x)
['start', 1, 2, 'hello', 3]
```

`insert` understands list indices as discussed in the section on slice notation, but for most uses it's easiest to think of `list.insert(n, elem)` as meaning `insert elem` just before the *n*th element of list. `insert` is just a convenience method. Anything that can be done with `insert` can also be done using slice assignment; that is, `list.insert(n, elem)` is the same thing as `list[n:n] = [elem]` when `n` is nonnegative. Using `insert` makes for somewhat more readable code, and `insert` even handles negative indices:

```
>>> x = [1, 2, 3]
>>> x.insert(-1, "hello")
>>> print(x)
[1, 2, 'hello', 3]
```

The `del` statement is the preferred method of deleting list items or slices. It doesn't do anything that can't be done with slice assignment, but it's usually easier to remember and easier to read:

```
>>> x = ['a', 2, 'c', 7, 9, 11]
>>> del x[1]
>>> x
['a', 'c', 7, 9, 11]
>>> del x[:2]
>>> x
[7, 9, 11]
```

In general, `del list[n]` does the same thing as `list[n:n+1] = []`, whereas `del list[m:n]` does the same thing as `list[m:n] = []`.

The `remove` method isn't the converse of `insert`. Whereas `insert` inserts an element at a specified location, `remove` looks for the first instance of a given value in a list and removes that value from the list:

```
>>> x = [1, 2, 3, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 5]
>>> x.remove(3)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

If `remove` can't find anything to remove, it raises an error. You can catch this error using the exception-handling abilities of Python, or you can avoid the problem by using `in` to check for the presence of something in a list before attempting to remove it.

The `reverse` method is a more specialized list modification method. It efficiently reverses a list in place:

```
>>> x = [1, 3, 5, 6, 7]
>>> x.reverse()
>>> x
[7, 6, 5, 3, 1]
```

5.4 Sorting lists

Lists can be sorted using the built-in Python `sort` method:

```
>>> x = [3, 8, 4, 0, 2, 1]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4, 8]
```

This does an in-place sort—that is, it changes the list being sorted. To sort a list without changing the original list, make a copy of it first.

```
>>> x = [2, 4, 1, 3]
>>> y = x[:]
>>> y.sort()
>>> y
[1, 2, 3, 4]
>>> x
[2, 4, 1, 3]
```

Sorting works with strings, too:

```
>>> x = ["Life", "Is", "Enchanting"]
>>> x.sort()
>>> x
['Enchanting', 'Is', 'Life']
```

The `sort` method can sort just about anything, because Python can compare just about anything. But there is one caveat in sorting. The default key method used by `sort` requires that all items in the list be of comparable types. That means that using the `sort` method on a list containing both numbers and strings will raise an exception:

```
>>> x = [1, 2, 'hello', 3]
>>> x.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

On the other hand, we can sort a list of lists:

```
>>> x = [[3, 5], [2, 9], [2, 3], [4, 1], [3, 2]]
>>> x.sort()
>>> x
[[2, 3], [2, 9], [3, 2], [3, 5], [4, 1]]
```

According to the built-in Python rules for comparing complex objects, the sublists are sorted first by ascending first element and then by ascending second element.

`sort` is even more flexible than this—it's possible to use your own key function to determine how elements of a list are sorted.

5.4.1 Custom sorting

To use custom sorting, you need to be able to define functions, something we haven't talked about. In this section we'll also use the fact that `len(string)` returns the number of characters in a string. String operations are discussed more fully in chapter 6.

By default, `sort` uses built-in Python comparison functions to determine ordering, which is satisfactory for most purposes. There will be times, though, when you want to sort a list in a way that doesn't correspond to this default ordering. For example, let's say we wish to sort a list of words by the number of characters in each word, in contrast to the lexicographic sort that would normally be carried out by Python.

To do this, write a function that will return the value, or key, that we want to sort on, and use it with the `sort` method. That function in the context of `sort` is a function that takes one argument and returns the key or value the `sort` function is to use.

For our number-of-characters ordering, a suitable key function could be

```
def compare_num_of_chars(string1):
    return len(string1)
```

This key function is trivial. It passes the length of each string back to the `sort` method, rather than the strings themselves.

After you define the key function, using it is a matter of passing it to the `sort` method using the `key` keyword. Because functions are Python objects, they can be passed around like any other Python object. Here's a small program that illustrates the difference between a default sort and our custom sort:

```
>>> def compare_num_of_chars(string1):
...     return len(string1)
```

```
>>> word_list = ['Python', 'is', 'better', 'than', 'C']
>>> word_list.sort()
>>> print(word_list)
['C', 'Python', 'better', 'is', 'than']
>>> word_list = ['Python', 'is', 'better', 'than', 'C']
>>> word_list.sort(key=compare_num_of_chars)
>>> print(word_list)
['C', 'is', 'than', 'Python', 'better']
```

The first list is in lexicographic order (with uppercase coming before lowercase), and the second list is ordered by ascending number of characters.

Custom sorting is very useful, but if performance is critical, it may be slower than the default. Usually this impact is minimal, but if the key function is particularly complex, the effect may be more than desired, especially for sorts involving hundreds of thousands or millions of elements.

One particular place to avoid custom sorts is where you want to sort a list in descending, rather than ascending, order. In this case, use the `sort` method's `reverse` parameter set to `True`. If for some reason you don't want to do that, it's still better to sort the list normally and then use the `reverse` method to invert the order of the resulting list. These two operations together—the standard sort and the reverse—will still be much faster than a custom sort.

5.4.2 The sorted() function

Lists have a built-in method to sort themselves, but other iterables in Python, like the keys of a dictionary, for example, don't have a `sort` method. Python also has the built-in function `sorted()`, which returns a sorted list from any iterable. `sorted()` uses the same `key` and `reverse` parameters as the sort method:

```
>>> x = (4, 3, 1, 2)
>>> y = sorted(x)
>>> y
[1, 2, 3, 4]
```

5.5 Other common list operations

A number of other list methods are frequently useful, but they don't fall into any specific category.

5.5.1 List membership with the in operator

It's easy to test if a value is in a list using the `in` operator, which returns a Boolean value. You can also use the converse, the `not in` operator:

```
>>> 3 in [1, 3, 4, 5]
True
>>> 3 not in [1, 3, 4, 5]
False
>>> 3 in ["one", "two", "three"]
False
>>> 3 not in ["one", "two", "three"]
True
```

5.5.2 List concatenation with the + operator

To create a list by concatenating two existing lists, use the + (list concatenation) operator. This will leave the argument lists unchanged.

```
>>> z = [1, 2, 3] + [4, 5]
>>> z
[1, 2, 3, 4, 5]
```

5.5.3 List initialization with the * operator

Use the `*` operator to produce a list of a given size, which is initialized to a given value. This is a common operation for working with large lists whose size is known ahead of time. Although you can use `append` to add elements and automatically expand the list as needed, you obtain greater efficiency by using `*` to correctly size the list at the start of the program. A list that doesn't change in size doesn't incur any memory reallocation overhead:

```
>>> z = [None] * 4
>>> z
[None, None, None, None]
```

When used with lists in this manner, * (which in this context is called the *list multiplication operator*) replicates the given list the indicated number of times and joins all the copies to form a new list. This is the standard Python method for defining a list of a given size ahead of time. A list containing a single instance of `None` is commonly used in list multiplication, but the list can be anything:

```
>>> z = [3, 1] * 2
>>> z
[3, 1, 3, 1]
```

5.5.4 List minimum or maximum with min and max

You can use `min` and `max` to find the smallest and largest elements in a list. You'll probably use these mostly with numerical lists, but they can be used with lists containing any type of element. Trying to find the maximum or minimum object in a set of objects of different types causes an error if it doesn't make sense to compare those types:

```
>>> min([3, 7, 0, -2, 11])
-2
>>> max([4, "Hello", [1, 2]])
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    max([4, "Hello",[1, 2]])
TypeError: unorderable types: str() > int()
```

5.5.5 List search with index

If you wish to find where in a list a value can be found (rather than wanting to know only if the value is in the list), use the `index` method. It searches through a

list looking for a list element equivalent to a given value and returns the position of that list element:

```
>>> x = [1, 3, "five", 7, -2]
>>> x.index(7)
3
>>> x.index(5)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

Attempting to find the position of an element that doesn't exist in the list at all raises an error, as shown here. This can be handled in the same manner as the analogous error that can occur with the `remove` method (that is, by testing the list with `in` before using `index`).

5.5.6 List matches with count

`count` also searches through a list, looking for a given value, but it returns the number of times that value is found in the list rather than positional information:

```
>>> x = [1, 2, 2, 3, 5, 2, 5]
>>> x.count(2)
3
>>> x.count(5)
2
>>> x.count(4)
0
```

5.5.7 Summary of list operations

You can see that lists are very powerful data structures, with possibilities that go far beyond plain old arrays. List operations are so important in Python programming that it's worth laying them out for easy reference, as shown in table 5.1.

Table 5.1 List operations

List operation	Explanation	Example
`[]`	Creates an empty list	`x = []`
`len`	Returns the length of a list	`len(x)`
`append`	Adds a single element to the end of a list	`x.append('y')`
`insert`	Inserts a new element at a given position in the list	`x.insert(0, 'y')`
`del`	Removes a list element or slice	`del(x[0])`
`remove`	Searches for and removes a given value from a list	`x.remove('y')`
`reverse`	Reverses a list in place	`x.reverse()`

Table 5.1 List operations

List operation	Explanation	Example
`sort`	Sorts a list in place	`x.sort()`
`+`	Adds two lists together	`x1 + x2`
`*`	Replicates a list	`x = ['y'] * 3`
`min`	Returns the smallest element in a list	`min(x)`
`max`	Returns the largest element in a list	`max(x)`
`index`	Returns the position of a value in a list	`x.index['y']`
`count`	Counts the number of times a value occurs in a list	`x.count('y')`
`in`	Returns whether an item is in a list	`'y' in x`

Being familiar with these list operations will make your life as a Python coder much easier.

5.6 Nested lists and deep copies

This is another advanced topic that you may want to skip if you're just learning the language.

Lists can be nested. One application of this is to represent two-dimensional matrices. The members of these can be referred to using two-dimensional indices. Indices for these work as follows:

```
>>> m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
>>> m[0]
[0, 1, 2]
>>> m[0][1]
1
>>> m[2]
[20, 21, 22]
>>> m[2][2]
22
```

This mechanism scales to higher dimensions in the manner you would expect.

Most of the time, this is all you need to concern yourself with. But there is an issue with nested lists that you may run into. This is the result of the combination of the way variables refer to objects and the fact that some objects (such as lists) can be modified (they're mutable). An example is the best way to illustrate:

```
>>> nested = [0]
>>> original = [nested, 1]
>>> original
[[0], 1]
```

Figure 5.1 shows what this looks like.

The value in the nested list can now be changed using either the nested or the original variables:

```
>>> nested[0] = 'zero'
>>> original
[['zero'], 1]
>>> original[0][0] = 0
>>> nested
[0]
>>> original
[[0], 1]
```

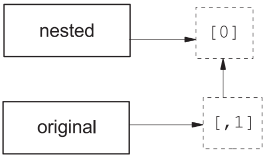


Figure 5.1 A list with its first item referring to a nested list

But if `nested` is set to another list, the connection between them is broken:

```
>>> nested = [2]
>>> original
[[0], 1]
```

Figure 5.2 illustrates this.

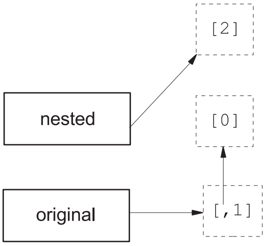


Figure 5.2 The first item of the original list is still a nested list, but the nested variable refers to a different list.

You've seen that you can obtain a copy of a list by taking a full slice (that is, `x[:]`). You can also obtain a copy of a list using the + or * operator (for example, `x + []` or `x * 1`). These are slightly less efficient than the slice method. All three create what is called a *shallow* copy of the list. This is probably what you want most of the time. But if your list has other lists nested in it, you may want to make a *deep* copy. You can do this with the `deepcopy` function of the `copy` module:

```
>>> original = [[0], 1]
>>> shallow = original[:]
>>> import copy
>>> deep = copy.deepcopy(original)
```

See figure 5.3 for an illustration.

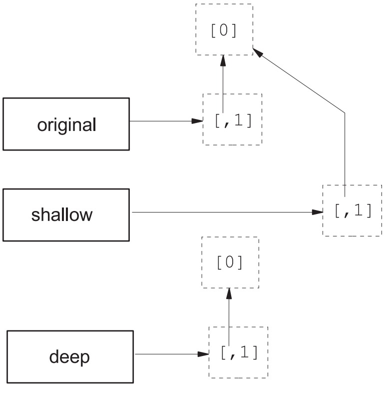


Figure 5.3 A shallow copy doesn't copy nested lists.

The lists pointed at by the original or shallow variables are connected. Changing the value in the nested list through either one of them affects the other:

```
>>> shallow[1] = 2
>>> shallow
[[0], 2]
>>> original
[[0], 1]
>>> shallow[0][0] = 'zero'
>>> original
[['zero'], 1]
```

The deep copy is independent of the original, and no change to it has any effect on the original list:

```
>>> deep[0][0] = 5
>>> deep
```

```
[[5], 1]
>>> original
[['zero'], 1]
```

This behavior is the same for any other nested objects in a list that are modifiable (such as dictionaries).

Now that you've seen what lists can do, it's time to look at tuples.

5.7 Tuples

Tuples are data structures that are very similar to lists, but they can't be modified. They can only be created. Tuples are so much like lists that you may wonder why Python bothers to include them. The reason is that tuples have important roles that can't be efficiently filled by lists, as keys for dictionaries.

5.7.1 Tuple basics

Creating a tuple is similar to creating a list: assign a sequence of values to a variable. A list is a sequence that is enclosed by `[` and `]`; a tuple is a sequence that is enclosed by `(` and `)`:

```
>>> x = ('a', 'b', 'c')
```

This line creates a three-element tuple.

After a tuple is created, using it is so much like using a list that it's easy to forget they're different data types:

```
>>> x[2]
'c'
>>> x[1:]
('b', 'c')
>>> len(x)
3
>>> max(x)
'c'
>>> min(x)
'a'
>>> 5 in x
False
>>> 5 not in x
True
```

The main difference between tuples and lists is that tuples are immutable. An attempt to modify a tuple results in a confusing error message, which is Python's way of saying it doesn't know how to set an item in a tuple.

```
>>> x[2] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You can create tuples from existing ones by using the `+` and `*` operators.

```
>>> x + x
('a', 'b', 'c', 'a', 'b', 'c')
```

```
>>> 2 * x
('a', 'b', 'c', 'a', 'b', 'c')
```

A copy of a tuple can be made in any of the same ways as for lists:

```
>>> x[:]
('a', 'b', 'c')
>>> x * 1
('a', 'b', 'c')
>>> x + ()
('a', 'b', 'c')
```

If you didn't read section 5.6, "Nested lists and deep copies," you can skip the rest of this paragraph. Tuples themselves can't be modified. But if they contain any mutable objects (for example, lists or dictionaries), these may be changed if they're still assigned to their own variables. Tuples that contain mutable objects aren't allowed as keys for dictionaries.

5.7.2 One-element tuples need a comma

A small syntactical point is associated with using tuples. Because the square brackets used to enclose a list aren't used elsewhere in Python, it's clear that `[]` means an empty list and `[1]` means a list with one element. The same thing isn't true with the parentheses used to enclose tuples. Parentheses can also be used to group items in expressions in order to force a certain evaluation order. If we say `(x + y)` in a Python program, do we mean that `x` and `y` should be added and then put into a one-element tuple, or do we mean that the parentheses should be used to force `x` and `y` to be added, before any expressions to either side come into play?

This is only a problem for tuples with one element, because tuples with more than one element always include commas to separate the elements, and the commas tell Python the parentheses indicate a tuple, not a grouping. In the case of one-element tuples, Python requires that the element in the tuple be followed by a comma, to disambiguate the situation. In the case of zero-element (empty) tuples, there's no problem. An empty set of parentheses must be a tuple, because it's meaningless otherwise.

```
>>> x = 3
>>> y = 4
>>> (x + y)   # This adds x and y.
7
>>> (x + y,)  # Including a comma indicates the parentheses denote a tuple.
(7,)
>>> ()        # To create an empty tuple, use an empty pair of parentheses.
()
```

5.7.3 Packing and unpacking tuples

As a convenience, Python permits tuples to appear on the left-hand side of an assignment operator, in which case variables in the tuple receive the corresponding values

from the tuple on the right-hand side of the assignment operator. Here's a simple example:

```
>>> (one, two, three, four) =  (1, 2, 3, 4)
>>> one
1
>>> two
2
```

This can be written even more simply, because Python recognizes tuples in an assignment context even without the enclosing parentheses. The values on the right-hand side are packed into a tuple and then unpacked into the variables on the left-hand side:

```
one, two, three, four = 1, 2, 3, 4
```

One line of code has replaced the following four lines of code:

```
one = 1
two = 2
three = 3
four = 4
```

This is a convenient way to swap values between variables. Instead of saying

```
temp = var1
var1 = var2
var2 = temp
```

just say

```
var1, var2 = var2, var1
```

To make things even more convenient, Python 3 has an extended unpacking feature, allowing an element marked with a * to absorb any number elements not matching the other elements. Again, some examples will make this clearer:

```
>>> x = (1, 2, 3, 4)
>>> a, b, *c = x
>>> a, b, c
(1, 2, [3, 4])
>>> a, *b, c = x
>>> a, b, c
(1, [2, 3], 4)
>>> *a, b, c = x
>>> a, b, c
([1, 2], 3, 4)
>>> a, b, c, d, *e = x
>>> a, b, c, d, e
(1, 2, 3, 4, [])
```

Note that the starred element receives all the surplus items as a list, and that if there are no surplus elements, it receives an empty list.

Packing and unpacking can be performed using list delimiters as well:

```
>>> [a, b] = [1, 2]
>>> [c, d] = 3, 4
>>> [e, f] = (5, 6)
>>> (g, h) = 7, 8
>>> i, j = [9, 10]
>>> k, l = (11, 12)
>>> a
1
>>> [b, c, d]
[2, 3, 4]
>>> (e, f, g)
(5, 6, 7)
>>> h, i, j, k, l
(8, 9, 10, 11, 12)
```

5.7.4 Converting between lists and tuples

Tuples can be easily converted to lists with the `list` function (which takes any sequence as an argument and produces a new list with the same elements as the original sequence). Similarly, lists can be converted to tuples with the `tuple` function (which does the same thing but produces a new tuple instead of a new list):

```
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

As an interesting side note, `list` is a convenient way to break a string into characters:

```
>>> list("Hello")
['H', 'e', 'l', 'l', 'o']
```

This works because `list` (and `tuple`) apply to any Python sequence, and a string is just a sequence of characters. (Strings are discussed fully in the next chapter.)

5.8 Sets

A *set* in Python is an unordered collection of objects used in situations where membership and uniqueness in the set are main things you need to know about that object. Just as with dictionary keys (as you'll see in chapter 7), the items in a set must be immutable and hashable. This means that ints, floats, strings, and tuples can be members of a set, but lists, dictionaries, and sets themselves can't.

5.8.1 Set operations

In addition to the operations that apply to collections in general, like `in`, `len`, and being able to use a `for` loop to iterate over all of their elements, sets also have several set-specific operations:

```
>>> x = set([1, 2, 3, 1, 3, 5])     ❶
>>> x
{1, 2, 3, 5}     ❷
```

```
>>> x.add(6)
>>> x
{1, 2, 3, 5, 6}
>>> x.remove(5)
>>> x
{1, 2, 3, 6}
>>> 1 in x
True
>>> 4 in x
False
>>> y = set([1, 7, 8, 9])
>>> x | y
{1, 2, 3, 6, 7, 8, 9}
>>> x & y
{1}
>>> x ^ y
{2, 3, 6, 7, 8, 9}
>>>
```

You can create a set by using `set` on a sequence, like a list ❶. When a sequence is made into a set, duplicates are removed ❷. After creating a set using the `set` function, you can use `add` ❸ and `remove` ❹ to change the elements in the set. The `in` keyword is used to check for membership of an object in a set ❺. You can also use `|` ❻ to get the union, or combination, of two sets, `&` to get their intersection ❼, and `^` ❽ to find their symmetric difference—that is, elements that are in one set or the other but not both.

These examples aren't a complete listing of set operations but are enough to give you a good idea of how sets work. For more information, refer to the official Python documentation.

5.8.2 Frozensets

Because sets aren't immutable and hashable, they can't belong to other sets. To remedy that situation there is another set type, frozenset, which is just like a set but can't be changed after creation. Because frozensets are immutable and hashable, they can be members of other sets:

```
>>> x = set([1, 2, 3, 1, 3, 5])
>>> z = frozenset(x)
>>> z
frozenset({1, 2, 3, 5})
>>> z.add(6)
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    z.add(6)
AttributeError: 'frozenset' object has no attribute 'add'
>>> x.add(z)
>>> x
{1, 2, 3, 5, frozenset({1, 2, 3, 5})}
```

5.9 Summary

Lists are a basic and highly useful data structure built into the Python language. In addition to demonstrating fairly standard array-like behavior, lists possess additional functionality, such as automatic resizing, the ability to use slice notation, and a good set of convenience functions, methods, and operators. Note that there are a few more list methods than were covered in this chapter. You'll find details on these in the appendix.

Tuples are similar to lists but can't be modified. They take up slightly less memory and are faster to access. They aren't as flexible but are more efficient than lists. Their normal use (from the point of view of a Python coder) is to serve as dictionary keys, which is discussed in chapter 7.

Sets are also sequence structures but with two differences from lists and tuples. Although sets do have an order, that order is arbitrary and not under the programmer's control. The second difference is that the elements in a set must be unique.

Lists, tuples, and sets are structures that embody the idea of a sequence of elements. As you'll see in the next chapter, strings are also sequences with some additional methods.

6 Strings

This chapter covers

- Understanding strings as sequences of characters
- Using basic string operations
- Inserting special characters and escape sequences
- Converting from objects to strings
- Formatting strings
- Using the byte type

Handling text—from user input, to filenames, to processing chunks of text—is a common chore in programming. Python comes with powerful tools to handle and format text. This chapter discusses the standard string and string-related operations in Python.

6.1 Strings as sequences of characters

For the purposes of extracting characters and substrings, strings can be considered sequences of characters, which means you can use index or slice notation:

```
>>> x = "Hello"
>>> x[0]
'H'
```

```
>>> x[-1]
'o'
>>> x[1:]
'ello'
```

One use for slice notation with strings is to chop the newline off the end of a string, usually a line that's just been read from a file:

```
>>> x = "Goodbye\n"
>>> x = x[:-1]
>>> x
'Goodbye'
```

This is just an example—you should know that Python strings have other, better methods to strip unwanted characters, but this illustrates the usefulness of slicing.

You can also determine how many characters are in the string by using the `len` function, just like finding out the number of elements in a list:

```
>>> len("Goodbye")
7
```

But strings aren't lists of characters. The most noticeable difference between strings and lists is that, unlike lists, *strings can't be modified*. Attempting to say something like `string.append('c')` or `string[0] = 'H'` will result in an error. You'll notice in the previous example that we stripped off the newline from the string by creating a string that was a slice of the previous one, not by modifying the previous string directly. This is a basic Python restriction, imposed for efficiency reasons.

6.2 Basic string operations

The simplest (and probably most common) way of combining Python strings is to use the string concatenation operator `+`:

```
>>> x = "Hello " + "World"
>>> x
'Hello World'
```

There is an analogous string multiplication operator that I have found sometimes, but not often, useful:

```
>>> 8 * "x"
'xxxxxxxx'
```

6.3 Special characters and escape sequences

You've already seen a few of the character sequences Python regards as special when used within strings: `\n` represents the newline character and `\t` represents the tab character. Sequences of characters that start with a backslash and that are used to represent other characters are called *escape sequences*. Escape sequences are generally used to represent *special characters*—that is, characters (such as tab and newline) that don't have a standard one-character printable representation. This section covers escape sequences, special characters, and related topics in more detail.

6.3.1 Basic escape sequences

Python provides a brief list of two-character escape sequences to use in strings (table 6.1).

Table 6.1 Escape sequences

Escape sequence	Character represented
`\'`	Single-quote character
`\"`	Double-quote character
`\\`	Backslash character
`\a`	Bell character
`\b`	Backspace character
`\f`	Formfeed character
`\n`	Newline character
`\r`	Carriage return character (not the same as `\n`)
`\t`	Tab character
`\v`	Vertical tab character

The ASCII character set, which is the character set used by Python and the standard character set on almost all computers, defines quite a few more special characters. They're accessed by the numeric escape sequences, described in the next section.

6.3.2 Numeric (octal and hexadecimal) and Unicode escape sequences

You can include any ASCII character in a string by using an octal (base 8) or hexadecimal (base 16) escape sequence corresponding to that character. An octal escape sequence is a backslash followed by three digits defining an octal number; the ASCII character corresponding to this octal number is substituted for the octal escape sequence. A hexadecimal escape sequence is similar but starts with \x rather than just \ and can consist of any number of hexadecimal digits. The escape sequence is terminated when a character is found that's not a hexadecimal digit. For example, in the ASCII character table, the character *m* happens to have decimal value 109. This is octal value 155 and hexadecimal value 6D, so:

```
>>> 'm'
'm'
>>> '\155'
'm'
>>> '\x6D'
'm'
```

All three expressions evaluate to a string containing the single character *m*. But these forms can also be used to represent characters that have no printable representation. The newline character \n, for example, has octal value 012 and hexadecimal value 0A:

```
>>> '\n'
'\n'
```

```
>>> '\012'
'\n'
>>> '\x0A'
'\n'
```

Because all strings in Python 3 are Unicode strings, they can also contain almost every character from every language available. Although a discussion of the Unicode system is far beyond this book, the following examples illustrate that you can also escape any Unicode character, either by number similar to that shown earlier or by Unicode name:

```
>>> unicode_a ='\N{LATIN SMALL LETTER A}'                    <- Escapes by
>>> unicode_a                                                    Unicode name
'a'                       <- 1
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'  <-
>>> unicode_a_with_acute
'á'
>>> "\u00E1"              <- Escapes by number,
'á'                          using \u
>>>
```

The Unicode character set includes the common ASCII characters (1).

6.3.3 *Printing vs. evaluating strings with special characters*

We talked before about the difference between evaluating a Python expression interactively and printing the result of the same expression using the `print` function. Although the same string is involved, the two operations can produce screen outputs that look different. A string that is evaluated at the top level of an interactive Python session will be shown with all of its special characters as octal escape sequences, which makes clear what is in the string. Meanwhile, the `print` function passes the string directly to the terminal program, which may interpret special characters in special ways. For example, here's what happens with a string consisting of an `a` followed by a newline, a tab, and a `b`:

```
>>> 'a\n\tb'
'a\n\tb'
>>> print('a\n\tb')
a
    b
```

In the first case, the newline and tab are shown explicitly in the string; in the second, they're used as newline and tab characters.

A normal `print` function also adds a newline to the end of the string. Sometimes (that is, when you have lines from files that already end with newlines) you may not want this behavior. Giving the `print` function an `end` parameter of `""` causes the `print` function to not append the newline:

```
>>> print("abc\n")
abc

>>> print("abc\n", end="")
abc
>>>
```

6.4 String methods

Most of the Python string methods are built into the standard Python string class, so all string objects have them automatically. The standard `string` module also contains some useful constants. Modules will be discussed in detail in chapter 10.

For the purposes of this section, you need only remember that most string methods are attached to the string object they operate on by a dot (`.`), as in `x.upper()`. That is, they're prepended with the string object followed by a dot.

Because strings are immutable, the string methods are used only to obtain their return value and don't modify the string object they're attached to in any way.

We'll begin with those string operations that are the most useful and commonly used and then go on to discuss some less commonly used but still useful operations. At the end, we'll discuss a few miscellaneous points related to strings. Not all of the string methods are documented here. See the documentation for a complete list of string methods.

6.4.1 The split and join string methods

Anyone who works with strings is almost certain to find the `split` and `join` methods invaluable. They're the inverse of one another—`split` returns a list of substrings in the string, and `join` takes a list of strings and puts them together to form a single string with the original string between each element. Typically, `split` uses whitespace as the delimiter to the strings it's splitting, but you can change that via an optional argument.

String concatenation using + is useful but not efficient for joining large numbers of strings into a single string, because each time + is applied, a new string object is created. Our previous "Hello World" example produced two string objects, one of which was immediately discarded. A better option is to use the `join` function:

```
>>> " ".join(["join", "puts", "spaces", "between", "elements"])
'join puts spaces between elements'
```

By changing the string used to `join`, you can put anything you want between the joined strings:

```
>>> "::".join(["Separated", "with", "colons"])
'Separated::with::colons'
```

You can even use an empty string, `""`, to join elements in a list:

```
>>> "".join(["Separated", "by", "nothing"])
'Separatedbynothing'
```

The most common use of `split` is probably as a simple parsing mechanism for string-delimited records stored in text files. By default, `split` splits on any whitespace, not just a single space character, but you can also tell it to split on a particular sequence by passing it an optional argument:

```
>>> x = "You\t\t can have tabs\t\n \t and newlines \n\n " \
               "mixed in"
>>> x.split()
['You', 'can', 'have', 'tabs', 'and', 'newlines', 'mixed', 'in']
>>> x = "Mississippi"
```

```
>>> x.split("ss")
['Mi', 'i', 'ippi']
```

Sometimes it's useful to permit the last field in a joined string to contain arbitrary text, including, perhaps, substrings that may match what `split` splits on when reading in that data. You can do this by specifying how many splits `split` should perform when it's generating its result, via an optional second argument. If you specify *n* splits, then `split` will go along the input string until it has performed *n* splits (generating a list with *n*+1 substrings as elements) or until it runs out of string. Here are some examples:

```
>>> x = 'a b c d'
>>> x.split(' ', 1)
['a', 'b c d']
>>> x.split(' ', 2)
['a', 'b', 'c d']
>>> x.split(' ', 9)
['a', 'b', 'c', 'd']
```

When using `split` with its optional second argument, you must supply a first argument. To get it to split on runs of whitespace while using the second argument, use `None` as the first argument.

I use `split` and `join` extensively, usually when working with text files generated by other programs. But you should know that if you're able to define your own data file format for use solely by your Python programs, there's a much better alternative to storing data in text files. We'll discuss it in chapter 13 when we talk about the `Pickle` module.

6.4.2 Converting strings to numbers

You can use the functions `int` and `float` to convert strings into integer or floating-point numbers, respectively. If they're passed a string that can't be interpreted as a number of the given type, they will raise a `ValueError` exception. Exceptions are explained in chapter 14, "Reading and writing files." In addition, you may pass `int` an optional second argument, specifying the numeric base to use when interpreting the input string:

```
>>> float('123.456')
123.456
>>> float('xxyy')
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: invalid literal for float(): xxyy
>>> int('3333')
3333
>>> int('123.456')                    <-- Can't have decimal point in integer
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 10: 123.456
>>> int('10000', 8)                   <-- Interprets 10000 as octal number
4096
>>> int('101', 2)
5
>>> int('ff', 16)
```

```
255
>>> int('123456', 6)
Traceback (innermost last):
 File "<stdin>", line 1, in ?
ValueError: invalid literal for int() with base 6: '123456'
```

Can't interpret 123456 as base 6 number

Did you catch the reason for that last error? We requested that the string be interpreted as a base 6 number, but the digit 6 can never appear in a base 6 number. Sneaky!

6.4.3 *Getting rid of extra whitespace*

A trio of simple methods that are surprisingly useful are the `strip`, `lstrip`, and `rstrip` functions. `strip` returns a new string that's the same as the original string, except that any whitespace at the *beginning or end* of the string has been removed. `lstrip` and `rstrip` work similarly, except that they remove whitespace only at the left or right end of the original string, respectively:

```
>>> x = "  Hello,    World\t\t "
>>> x.strip()
'Hello,    World'
>>> x.lstrip()
'Hello,    World\t\t '
>>> x.rstrip()
'  Hello,    World'
```

In this example, tab characters are considered to be whitespace. The exact meaning may differ across operating systems, but you can always find out what Python considers to be whitespace by accessing the `string.whitespace` constant. On my Windows system, it gives the following:

```
import string
>>> string.whitespace
' \t\n\r\x0b\x0c'
>>> " \t\n\r\v\f"
' \t\n\r\x0b\x0c'
```

The characters given in backslashed hex (\xnn) format represent the vertical tab and formfeed characters. The space character is in there as itself. It may be tempting to change the value of this variable, to attempt to affect how `strip` and so forth work, but don't do it. Such an action isn't guaranteed to give you the results you're looking for.

But you can change which characters `strip`, `rstrip`, and `lstrip` remove by passing a string containing the characters to be removed as an extra parameter:

```
>>> x = "www.python.org"
>>> x.strip("w")
'.python.org'
>>> x.strip("gor")
'www.python.'
>>> x.strip(".gorw")
'python'
```

Strips off all *w*s

❶ Strips off all *g*s, *o*s, and *r*s

Strips of all dots, *g*s, *o*s, *r*s and *w*s

Note that `strip` removes any and all of the characters in the extra parameter string, no matter in which order they occur ❶.

The most common use for these functions is as a quick way of cleaning up strings that have just been read in. This is particularly helpful when you're reading lines from files (discussed in chapter 13), because Python always reads in an entire line, including the trailing newline, if it exists. When you get around to processing the line read in, you typically don't want the trailing newline. `rstrip` is a convenient way to get rid of it.

6.4.4 String searching

The string objects provide a number of methods to perform simple string searches. Before I describe them, though, let's talk about another module in Python: `re`. (This module will be discussed in depth in chapter 17, "Regular expressions.")

Another method for searching strings: the re module

The `re` module also does string searching but in a far more flexible manner, using *regular expressions*. Rather than searching for a single specified substring, an `re` search can look for a string pattern. You could look for substrings that consist entirely of digits, for example.

Why am I mentioning this, when `re` is discussed fully later? In my experience, many uses of basic string searches are inappropriate. You'd benefit from a more powerful searching mechanism but aren't aware that one exists, and so you don't even look for something better. Perhaps you have an urgent project involving strings and don't have time to read this entire book. If basic string searching will do the job for you, that's great. But be aware that you have a more powerful alternative.

The four basic string-searching methods are all similar: `find`, `rfind`, `index`, and `rindex`. A related method, `count`, counts how many times a substring can be found in another string. We'll describe `find` in detail and then examine how the other methods differ from it.

`find` takes one required argument: the substring being searched for. `find` returns the position of the first character of the first instance of `substring` in the `string` object, or `-1` if `substring` doesn't occur in the string:

```
>>> x = "Mississippi"
>>> x.find("ss")
2
>>> x.find("zz")
-1
```

`find` can also take one or two additional, optional arguments. The first of these, if present, is an integer `start`; it causes `find` to ignore all characters before position `start` in `string` when searching for `substring`. The second optional argument, if present, is an integer `end`; it causes `find` to ignore characters at or after position `end` in `string`:

```
>>> x = "Mississippi"
>>> x.find("ss", 3)
5
```

```
>>> x.find("ss", 0, 3)
-1
```

`rfind` is almost the same as `find`, except that it starts its search at the end of `string` and so returns the position of the first character of the last occurrence of `substring` in `string`:

```
>>> x = "Mississippi"
>>> x.rfind("ss")
5
```

`rfind` can also take one or two optional arguments, with the same meanings as those for `find`.

`index` and `rindex` are identical to `find` and `rfind`, respectively, except for one difference: if `index` or `rindex` fails to find an occurrence of `substring` in `string`, it doesn't return `-1` but rather raises a `ValueError` exception. Exactly what this means will be clear after you read chapter 14, "Exceptions."

`count` is used identically to any of the previous four functions but returns the number of non-overlapping times the given substring occurs in the given string:

```
>>> x = "Mississippi"
>>> x.count("ss")
2
```

You can use two other string methods to search strings: `startswith` and `endswith`. These methods return a `True` or `False` result depending on whether the string they're used on starts or ends with one of the strings given as parameters:

```
>>> x = "Mississippi"
>>> x.startswith("Miss")
True
>>> x.startswith("Mist")
False
>>> x.endswith("pi")
True
>>> x.endswith("p")
False
```

Both `startswith` and `endswith` can look for more than one string at a time. If the parameter is a tuple of strings, both methods check for all of the strings in the tuple and return a `True` if any one of them is found:

```
>>> x.endswith(("i", "u"))
True
```

`startswith` and `endswith` are useful for simple searches.

6.4.5 Modifying strings

Strings are immutable, but string objects have a number of methods that can operate on that string and return a new string that's a modified version of the original string. This provides much the same effect as direct modification for most purposes. You can find a more complete description of these methods in the documentation.

You can use the `replace` method to replace occurrences of `substring` (its first argument) in the string with `newstring` (its second argument). It also takes an optional third argument (see the documentation for details):

```
>>> x = "Mississippi"
>>> x.replace("ss", "+++")
'Mi+++i+++ippi'
```

As with the string search functions, the `re` module provides a much more powerful method of substring replacement.

The functions `string.maketrans` and `string.translate` may be used together to translate characters in strings into different characters. Although rarely used, these functions can simplify your life when they're needed.

Let's say, for example, that you're working on a program that translates string expressions from one computer language into another. The first language uses `~` to mean logical not, whereas the second language uses `!`; the first language uses `^` to mean logical and, whereas the second language uses `&`; the first language uses `(` and `)`, where the second language uses `[` and `]`. In a given string expression, you need to change all instances of `~` to `!`, all instances of `^` to `&`, all instances of `(` to `[`, and all instances of `)` to `]`. You could do this using multiple invocations of `replace`, but an easier and more efficient way is

```
>>> x = "~x ^ (y % z)"
>>> table = x.maketrans("~^()", "!&[]")
>>> x.translate(table)
'!x & [y % z]'
```

The first line uses `maketrans` to make up a translation table from its two string arguments. The two arguments must each contain the same number of characters, and a table will be made such that looking up the *n*th character of the first argument in that table gives back the *n*th character of the second argument.

Next, the table produced by `maketrans` is passed to `translate`. Then, `translate` goes over each of the characters in its `string` object and checks to see if they can be found in the table given as the second argument. If a character can be found in the translation table, `translate` replaces that character with the corresponding character looked up in the table, to produce the translated string.

You can give an optional argument to `translate`, to specify characters that should be removed from the string entirely. See the documentation for details.

Other functions in the `string` module perform more specialized tasks. `string.lower` converts all alphabetic characters in a string to lowercase, and `upper` does the opposite. `capitalize` capitalizes the first character of a string, and `title` capitalizes all words in a string. `swapcase` converts lowercase characters to uppercase and uppercase to lowercase in the same string. `expandtabs` gets rid of tab characters in a string by replacing each tab with a specified number of spaces. `ljust`, `rjust`, and `center` pad a string with spaces, to justify it in a certain field width. `zfill` left-pads a numeric string with zeros. Refer to the documentation for details of these methods.

6.4.6 Modifying strings with list manipulations

Because strings are immutable objects, there's no way to directly manipulate them in the same way you can lists. Although the operations that operate on strings to produce new strings (leaving the original strings unchanged) are useful for many things, sometimes you want to be able to manipulate a string as if it were a list of characters. In that case, just turn it into a list of characters, do whatever you want, and turn the resulting list back into a string:

```
>>> text = "Hello, World"
>>> wordList = list(text)
>>> wordList[6:] = []                    ← Removes everything after comma
>>> wordList.reverse()
>>> text = "".join(wordList)             ← Joins with no space between
>>> print(text)
,olleH
```

Although you can use `split` to turn your string into a list of characters, the type-conversion function `list` is easier to use and to remember (and, for what it's worth, you can turn a string into a tuple of characters using the built-in `tuple` function). To turn the list back into a string, use `"".join`.

You shouldn't go overboard with this method because it causes the creation and destruction of new `string` objects, which is relatively expensive. Processing hundreds or thousands of strings in this manner probably won't have much of an impact on your program. Processing millions probably will.

6.4.7 Useful methods and constants

`string` objects also have several useful methods to report qualities of the string, whether it consists of digits or alphabetic characters, is all uppercase or lowercase, and so on:

```
>>> x = "123"
>>> x.isdigit()
True
>>> x.isalpha()
False
>>> x = "M"
>>> x.islower()
False
>>> x.isupper()
True
```

For a fuller list of all the possible string methods, refer to the string section of the official Python documentation.

Finally, the `string` module defines some useful constants. You've already seen `string.whitespace`, which is a string made up of the characters Python thinks of as whitespace on your system. `string.digits` is the string `'0123456789'`. `string.hexdigits` includes all the characters in `string.digits`, as well as `'abcdefABCDEF'`, the extra characters used in hexadecimal numbers. `string.octdigits` contains

'01234567'—just those digits used in octal numbers. string.lowercase contains all lowercase alphabetic characters; string.uppercase contains all uppercase alphabetic characters; string.letters contains all of the characters in string.lowercase and string.uppercase. You might be tempted to try assigning to these constants to change the behavior of the language. Python would let you get away with this, but it would probably be a bad idea.

Remember that strings are sequences of characters, so you can use the convenient Python in operator to test for a character's membership in any of these strings, although usually the existing string methods is simpler and easier.

The most common string operations are shown in table 6.2.

Table 6.2 String operations

String operation	Explanation	Example
+	Adds two strings together	x = "hello " + "world"
*	Replicates a string	x = " " * 20
upper	Converts a string to uppercase	x.upper()
lower	Converts a string to lowercase	x.lower()
title	Capitalizes the first letter of each word in a string	x.title()
find, index	Searches for the target in a string	x.find(y) x.index(y)
rfind, rindex	Searches for the target in a string, from the end of the string	x.rfind(y) x.rindex(y)
startswith, endswith	Checks the beginning or end of a string for a match	x.startswith(y) x.endswith(y)
replace	Replaces the target with a new string	x.replace(y, z)
strip, rstrip, lstrip	Removes whitespace or other characters from the ends of a string	x.strip()
encode	Converts a Unicode string to a bytes object	x.encode("utf_8")

Note that these methods don't change the string itself but return either a location in the string or a new string.

6.5 Converting from objects to strings

In Python, almost anything can be converted to some sort of a string representation, using the built-in repr function. Lists are the only complex Python data types you're familiar with so far, so let's turn some lists into their representations:

```
>>> repr([1, 2, 3])
'[1, 2, 3]'
```

```
>>> x = [1]
>>> x.append(2)
>>> x.append([3, 4])
>>> 'the list x is ' + repr(x)
'the list x is [1, 2, [3, 4]]'
```

The example uses `repr` to convert the list `x` into a string representation, which is then concatenated with the other string to form the final string. Without the use of `repr`, this wouldn't work. In an expression like `"string" + [1, 2] + 3`, are you trying to add strings, or add lists, or just add numbers? Python doesn't know what you want in such a circumstance, and it will do the safe thing (raise an error) rather than make any assumptions. In the previous example, all the elements had to be converted to string representations before the string concatenation would work.

Lists are the only complex Python objects that have been described to this point, but `repr` can be used to obtain some sort of string representation for almost any Python object. To see this, try `repr` around a built-in complex object—an actual Python function:

```
>>> repr(len)
'<built-in function len>'
```

Python hasn't produced a string containing the code that implements the `len` function, but it has at least returned a string—`<built-in function len>`—that describes what that function is. If you keep the `repr` function in mind and try it on each Python data type (dictionaries, tuples, classes, and the like) as we get to them in the book, you'll see that no matter what type of Python object you have, you can get a string saying something about that object.

This is great for debugging programs. If you're in doubt as to what's held in a variable at a certain point in your program, use `repr` and print out the contents of that variable.

We've covered how Python can convert any object into a string that describes that object. The truth is, Python can do this in either of two different ways. The `repr` function always returns what might be loosely called the *formal string representation* of a Python object. More specifically, `repr` returns a string representation of a Python object from which the original object can be rebuilt. For large, complex objects, this may not be the sort of thing you wish to see in debugging output or status reports.

Python also provides the built-in `str` function. In contrast to `repr`, `str` is intended to produce *printable* string representations, and it can be applied to any Python object. `str` returns what might be called the *informal string representation* of the object. A string returned by `str` need not define an object fully and is intended to be read by humans, not by Python code.

You won't notice any difference between `repr` and `str` when you first start using them, because until you begin using the object-oriented features of Python, there is no difference. `str` applied to any built-in Python object always calls `repr` to calculate its result. It's only when you start defining your own classes that the difference between `str` and `repr` becomes important. This will be discussed in chapter 15.

So why talk about this now? Basically, I wanted you to be aware that there's more going on behind the scenes with `repr` than being able to easily write `print` functions for debugging. As a matter of good style, you may want to get into the habit of using `str` rather than `repr` when creating strings for displaying information.

6.6 Using the format method

You can format strings in Python 3 in two ways. The newer way to format strings in Python is to use the string class's `format` method. The `format` method combines a format string containing replacement fields marked with { } with replacement values taken from the parameters given to the `format` command. If you need to include a literal { or } in the string, you double it to {{ or }}. The `format` command is a powerful string-formatting mini-language and offers almost endless possibilities for manipulating string formatting. On the other hand, it's fairly simple to use for the most common use cases, so we'll look at a few basic patterns. Then, if you need to use the more advanced options, you can refer to the string-formatting section of the standard library documentation.

6.6.1 The format method and positional parameters

The simplest use of the string `format` method uses numbered replacement fields that correspond to the parameters passed to the `format` function:

```
>>> "{0} is the {1} of {2}".format("Ambrosia", "food", "the gods")    ❶
'Ambrosia is the food of the gods'
>>> "{{Ambrosia}} is the {0} of {1}".format("food", "the gods")       ❷
'{Ambrosia} is the food of the gods'
```

Note that the `format` method is applied to the format string, which can also be a string variable ❶. Doubling the { } characters escapes them so that they don't mark a replacement field ❷.

This example has three replacement fields, `{0}`, `{1}`, and `{2}`, which are in turn filled by the first, second, and third parameters. No matter where in the format string we place `{0}`, it will always be replaced by the first parameter, and so on.

You can also use the positional parameters.

6.6.2 The format method and named parameters

The `format` method also recognizes named parameters and replacement fields:

```
>>> "{food} is the food of {user}".format(food="Ambrosia",
...         user="the gods")
'Ambrosia is the food of the gods'
```

In this case, the replacement parameter is chosen by matching the name of the replacement field to the name of the parameter given to the `format` command.

You can also use both positional and named parameters, and you can even access attributes and elements within those parameters:

```
>>> "{0} is the food of {user[1]}".format("Ambrosia",
```

```
...            user=["men", "the gods", "others"])
'Ambrosia is the food of the gods'
```

In this case, the first parameter is positional, and the second, `user[1]`, refers to the second element of the named parameter `user`.

6.6.3 Format specifiers

Format specifiers let you specify the result of the formatting with even more power and control than the formatting sequences of the older style of string formatting. The format specifier lets you control the fill character, alignment, sign, width, precision, and type of the data when it's substituted for the replacement field. As noted earlier, the syntax of format specifiers is a mini-language in its own right and too complex to cover completely here, but the following examples give you an idea of its usefulness:

```
>>> "{0:10} is the food of gods".format("Ambrosia")                    ❶
'Ambrosia   is the food of gods'
>>> "{0:{1}} is the food of gods".format("Ambrosia", 10)               ❷
'Ambrosia   is the food of gods'
>>> "{food:{width}} is the food of gods".format(food="Ambrosia", width=10)
'Ambrosia   is the food of gods'
>>> "{0:>10} is the food of gods".format("Ambrosia")                   ❸
'  Ambrosia is the food of gods'
>>> "{0:&>10} is the food of gods".format("Ambrosia")                  ❹
'&&Ambrosia is the food of gods'
```

`:10` is a format specifier that makes the field 10 spaces wide and pads with spaces ❶. `:{1}` takes the width from the second parameter ❷. `:>10` forces left justification of the field and pads with spaces ❸. `:&>10` forces left justification and pads with `&` instead of spaces ❹ .

6.7 Formatting strings with %

This section covers formatting strings with the *string modulus* (`%`) operator. It's used to combine Python values into formatted strings for printing or other use. C users will notice a strange similarity to the `printf` family of functions. The use of `%` for string formatting is the old style of string formatting, and I cover it here because it was the standard in earlier versions of Python and you're likely to see it in code that's been ported from earlier versions of Python or was written by coders familiar with those versions. This style of formatting shouldn't be used in new code, because it's slated to be deprecated and then removed from the language in the future.

Here's an example:

```
>>> "%s is the %s of %s" % ("Ambrosia", "food", "the gods")
'Ambrosia is the food of the gods'
```

The string modulus operator (the bold `%` that occurs in the middle, not the three instances of `%s` that come before it in the example) takes two parts: the left side, which is a string; and the right side, which is a tuple. The string modulus operator scans the left string for special *formatting sequences* and produces a new string by substituting the

values on the right side for those formatting sequences, in order. In this example, the only formatting sequences on the left side are the three instances of `%s`, which stands for "stick a string in here."

Passing in different values on the right side produces different strings:

```
>>> "%s is the %s of %s" % ("Nectar", "drink", "gods")
'Nectar is the drink of gods'
>>> "%s is the %s of the %s" % ("Brussels Sprouts", "food",
...        "foolish")
'Brussels Sprouts is the food of the foolish'
```

The members of the tuple on the right will have `str` applied to them automatically by `%s`, so they don't have to already be strings:

```
>>> x = [1, 2, "three"]
>>> "The %s contains: %s" % ("list", x)
"The list contains: [1, 2, 'three']"
```

6.7.1 *Using formatting sequences*

All formatting sequences are substrings contained in the string on the left side of the central `%`. Each formatting sequence begins with a percent sign and is followed by one or more characters that specify what is to be substituted for the formatting sequence and how the substitution is accomplished. The `%s` formatting sequence used previously is the simplest formatting sequence, and it indicates that the corresponding string from the tuple on the right side of the central `%` should be substituted in place of the `%s`.

Other formatting sequences can be more complex. This one specifies the field width (total number of characters) of a printed number to be six, specifies the number of characters after the decimal point to be two, and left-justifies the number in its field. I've put in angle brackets so you can see where extra spaces are inserted into the formatted string:

```
>>> "Pi is <%-6.2f>" % 3.14159 # use of the formatting sequence: %-6.2f
'Pi is <3.14  >'
```

All the options for characters that are allowable in formatting sequences are given in the documentation. There are quite a few options, but none are particularly difficult to use. Remember, you can always try a formatting sequence interactively in Python to see if it does what you expect it to do.

6.7.2 *Named parameters and formatting sequences*

Finally, one additional feature is available with the `%` operator that can be useful in certain circumstances. Unfortunately, to describe it we're going to have to employ a Python feature we haven't used yet—dictionaries, commonly called *hashtables* or *associative arrays* by other languages. You can skip ahead to the next chapter, "Dictionaries," to learn about dictionaries, skip this section for now and come back to it later, or read straight through, trusting to the examples to make things clear.

Formatting sequences can specify what should be substituted for them by name rather than by position. When you do this, each formatting sequence has a name in parentheses, immediately following the initial % of the formatting sequence, like so:

```
"%(pi).2f"
```

Note name in parentheses

In addition, the argument to the right of the % operator is no longer given as a single value or tuple of values to be printed but rather as a dictionary of values to be printed, with each named formatting sequence having a correspondingly named key in the dictionary. Using the previous formatting sequence with the string modulus operator, we might produce code like this:

```
>>> num_dict = {'e': 2.718, 'pi': 3.14159}
>>> print("%(pi).2f - %(pi).4f - %(e).2f" % num_dict)
3.14 - 3.1416 - 2.72
```

This is particularly useful when you're using format strings that perform a large number of substitutions, because you no longer have to keep track of the positional correspondences of the right-side tuple of elements with the formatting sequences in the format string. The order in which elements are defined in the `dict` argument is irrelevant, and the template string may use values from `dict` more than once (as it does with the `'pi'` entry).

Controlling output with the print function

Python's built-in `print` function also has some options that can make handling simple string output easier. When used with one parameter, `print` prints the value and a newline character, so that a series of calls to `print` print each value on a separate line:

```
>>> print("a")
a
>>> print("b")
b
```

But `print` can do more than that. You can also give the `print` function a number of arguments, and they will be printed on the same line, separated by a space and ending with a newline:

```
>>> print("a", "b", "c")
a b c
```

If that's not quite what you need, you can give the `print` function additional parameters to control what separates each item and what ends the line:

```
>>> print("a", "b", "c", sep="|")
a|b|c
>>> print("a", "b", "c", end="\n\n")
a b c

>>>
```

In chapter 12, you'll also see that the `print` function can be used to print to files as well as console output.

Using the `print` function's options gives you enough control for simple text output, but more complex situations are best served by using the `format` method.

6.8 Bytes

A `bytes` object is similar to a `string` object but with an important difference. A `string` is an immutable sequence of Unicode characters, whereas a `bytes` object is a sequence of integers with values from 0 to 256. Bytes can be necessary when you're dealing with binary data—for example, reading from a binary data file.

The key thing to remember is that `bytes` objects may look like strings, but they can't be used exactly like a string and they can't be combined with strings:

```
>>> unicode_a_with_acute = '\N{LATIN SMALL LETTER A WITH ACUTE}'
>>> unicode_a_with_acute
'á'
>>> xb = unicode_a_with_acute.encode()        ❶
>>> xb
b'\xc3\xa1'                ❷
>>> xb += 'A'                                  ❸
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    xb += 'A'
TypeError: can't concat bytes to str
>>> xb.decode()                                ❹
'á'
```

The first thing you can see is that to convert from a regular (Unicode) string to `bytes`, you need to call the string's `encode` method ❶. After it's encoded to a `bytes` object, the character is now 2 bytes and no longer prints the same way the string did ❷. Further, if you attempt to add a `bytes` object and a string object together, you get a type error, because the two are incompatible types ❸. Finally, to convert a `bytes` object back to a string, you need to call that object's `decode` method ❹.

Most of the time, you shouldn't need to think about Unicode or bytes at all. But when you need to deal with international character sets, an increasingly common issue, you must understand the difference between regular strings and `bytes`.

6.9 Summary

Python's string type gives you several powerful tools for text processing. Almost all of those tools are the methods attached to any `string` object, although there's an even more powerful set of tools in the `re` module. The standard string methods can search and replace, trim off extra characters, change case, and much more. Because strings are immutable—that is, they can't be changed—the operations that "change" strings return a copy with the changes, but the original remains untouched.

After lists and strings, the next important Python data structure to consider is the dictionary, before we move on to control structures.

7 Dictionaries

This chapter covers

- Defining a dictionary
- Using dictionary operations
- Determining what can be used as a key
- Creating sparse matrices
- Using dictionaries as caches
- Trusting the efficiency of dictionaries

This chapter discusses dictionaries, Python's name for associative arrays, which it implements using hash tables. Dictionaries are amazingly useful, even in simple programs.

Because dictionaries are less familiar to many programmers than other basic data structures such as lists and strings, some of the examples illustrating dictionary use are slightly more complex than the corresponding examples for other built-in data structures. It may be necessary to read parts of the next chapter, "Control flow," to fully understand some of the examples in this chapter.

7.1 What is a dictionary?

If you've never used associative arrays or hash tables in other languages, then a good way to start understanding the use of dictionaries is to compare them with lists:

- Values in lists are accessed by means of integers called *indices*, which indicate where in the list a given value is found.
- Dictionaries access values by means of integers, strings, or other Python objects called *keys*, which indicate where in the dictionary a given value is found. In other words, both lists and dictionaries provide indexed access to arbitrary values, but the set of items that can be used as dictionary indices is much larger than, and contains, the set of items that can be used as list indices. Also, the mechanism that dictionaries use to provide indexed access is quite different than that used by lists.
- Both lists and dictionaries can store objects of any type.
- Values stored in a list are implicitly *ordered* by their position in the list, because the indices that access these values are consecutive integers; you may or may not care about this ordering, but you can use it if desired. Values stored in a dictionary aren't implicitly ordered relative to one another because dictionary keys aren't just numbers. Note that if you're using a dictionary, you can define an ordering on the items in a dictionary by using another data structure (often a list) to store such an ordering explicitly; this doesn't change the fact that dictionaries have no implicit (built-in) ordering.

In spite of the differences between them, use of dictionaries and lists often appears alike. As a start, an empty dictionary is created much like an empty list, but with curly braces instead of square brackets:

```
>>> x = []
>>> y = {}
```

Here, the first line creates a new, empty list and assigns it to x. The second creates a new, empty dictionary, and assigns it to y.

After you create a dictionary, values may be stored in it as if it were a list:

```
>>> y[0] = 'Hello'
>>> y[1] = 'Goodbye'
```

Even in these assignments, there is already a significant operational difference between the dictionary and list usage. Trying to do the same thing with a list would result in an error, because in Python it's illegal to assign to a position in a list that doesn't already exist. For example, if we try to assign to the 0th element of the list x, we receive an error:

```
>>> x[0] = 'Hello'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

This isn't a problem with dictionaries; new positions in dictionaries are created as necessary.

Having stored some values in the dictionary, we can now access and use them:

```
>>> print(y[0])
Hello
>>> y[1] + ", Friend."
'Goodbye, Friend.'
```

All in all, this makes a dictionary look pretty much like a list. Now for the big difference; let's store (and use) some values under keys that aren't integers:

```
>>> y["two"] = 2
>>> y["pi"] = 3.14
>>> y["two"] * y["pi"]
6.2800000000000002
```

This is definitely something that can't be done with lists! Whereas list indices must be integers, dictionary keys are much less restricted—they may be numbers, strings, or one of a wide range of other Python objects. This makes dictionaries a natural for jobs that lists can't do. For example, it makes more sense to implement a telephone directory application with dictionaries than with lists, because the phone number for a person can be stored indexed by that person's last name.

7.1.1 Why dictionaries are called dictionaries

A dictionary is a way of mapping from one set of arbitrary objects to an associated but equally arbitrary set of objects. Actual dictionaries, thesauri, or translation books are a good analogy in the real world. To see how natural this correspondence is, here is the start of an English-to-French color translator:

```
>>> english_to_french = {}
>>> english_to_french['red'] = 'rouge
>>> english_to_french['blue'] = 'bleu'
>>> english_to_french['green'] = 'vert'
>>> print("red is", english_to_french['red'])
red is rouge
```

Creates empty dictionary (line 1)

Stores three words in it (lines 2–4)

Obtains value for 'red' (line 5)

7.2 Other dictionary operations

Besides basic element assignment and access, dictionaries support a number of other operations. You can define a dictionary explicitly as a series of key/value pairs separated by commas:

```
>>> english_to_french = {'red': 'rouge', 'blue': 'bleu', 'green': 'vert'}
```

`len` returns the number of entries in a dictionary:

```
>>> len(english_to_french)
3
```

You can obtain all the keys in the dictionary with the keys method. This is often used to iterate over the contents of a dictionary using Python's for loop, described in chapter 8:

```
>>> list(english_to_french.keys())
['green', 'blue', 'red']
```

The order of the keys in a list returned by keys has no meaning—they aren't necessarily sorted, and they don't necessarily occur in the order they were created. Your Python may print out the keys in a different order than my Python did. If you need keys sorted, you can store them in a list variable and then sort that list.

It's also possible to obtain all the values stored in a dictionary, using values:

```
>>> list(english_to_french.values())
 ['vert', 'bleu', 'rouge']
```

This method isn't used nearly as often as keys.

You can use the items method to return all keys and their associated values as a sequence of tuples:

```
>>> list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
```

Like keys, this is often used in conjunction with a for loop to iterate over the contents of a dictionary.

The del statement can be used to remove an entry (key/value pair) from a dictionary:

```
>>> list(english_to_french.items())
[('green', 'vert'), ('blue', 'bleu'), ('red', 'rouge')]
>>> del english_to_french['green']
>>> list(english_to_french.items())
 [('blue', 'bleu'), ('red', 'rouge')]
```

Dictionary view objects

The keys, values, and items methods return not lists but rather *views* that behave like sequences but are dynamically updated whenever the dictionary changes. That's why we need to use the list function to make them appear as a list in our examples. Otherwise, they behave like sequences, allowing code to iterate over them in a for loop, use in to check membership in them, and so on.

The view returned by keys (and in some cases the view returned by items) also behaves like a set, with union, difference, and intersection operations.

Attempting to access a key that isn't in a dictionary is an error in Python. To handle this, you can test the dictionary for the presence of a key with the in keyword, which returns True if a dictionary has a value stored under the given key and False otherwise:

```
>>> 'red' in english_to_french
True
>>> 'orange' in english_to_french
False
```

Alternatively, you can use the get function. It returns the value associated with a key, if the dictionary contains that key, but returns its second argument if the dictionary doesn't contain the key:

```
>>> print(english_to_french.get('blue', 'No translation'))
bleu
>>> print(english_to_french.get('chartreuse', 'No translation'))
No translation
```

The second argument is optional. If it isn't included, get returns None if the dictionary doesn't contain the key.

Similarly, if you want to safely get a key's value *and* make sure it's set to a default in the dictionary, you can use the setdefault method:

```
>>> print(english_to_french.setdefault('chartreuse', 'No translation'))
No translation
```

The difference between get and setdefault is that after the setdefault call, there is a key in the dictionary 'chartreuse' with the value 'No translation'.

You can obtain a copy of a dictionary using the copy method:

```
>>> x = {0: 'zero', 1: 'one'}
>>> y = x.copy()
>>> y
{0: 'zero', 1: 'one'}
```

This makes a shallow copy of the dictionary. This will likely be all you need in most situations. For dictionaries that contain any modifiable objects as values (that is, lists or other dictionaries), you may want to make a deep copy using the copy.deepcopy function. See "Nested lists and deep copies" (section 5.6) of "Lists, tuples, and sets" (chapter 5) for an introduction to the concept of shallow and deep copies.

The update method updates a first dictionary with all the key/value pairs of a second dictionary. For keys that are common to both, the values from the second dictionary override those of the first:

```
>>> z = {1: 'One', 2: 'Two'}
>>> x = {0: 'zero', 1: 'one'}
>>> x.update(z)
>>> x
{0: 'zero', 1: 'One', 2: 'Two'}
```

Dictionary methods give you a full set of tools to manipulate and use dictionaries. For quick reference, table 7.1 contains some of the main dictionary functions in table format.

Table 7.1 Dictionary operations

Dictionary operation	Explanation	Example
`{}`	Creates an empty dictionary	`x = {}`
`len`	Returns the number of entries in a dictionary	`len(x)`
`keys`	Returns a view of all keys in a dictionary	`x.keys()`

Table 7.1 Dictionary operations *(continued)*

Dictionary operation	Explanation	Example
`values`	Returns a view of all values in a dictionary	`x.values()`
`items`	Returns a view of all items in a dictionary	`x.items()`
`del`	Removes an entry from a dictionary	`del(x[key])`
`in`	Tests whether a key exists in a dictionary	`'y' in x`
`get`	Returns the value of a key or a configurable default	`x.get('y',None)`
`setdefault`	Returns the value if the key is in the dictionary; otherwise, sets the value for the key to the default and returns the value	`x.setdefault('y', None)`
`copy`	Makes a copy of a dictionary	`y = x.copy()`
`update`	Combines the entries of two dictionaries	`x.update(z)`

This isn't a complete list of all dictionary operations. For a complete list, see the appendix or refer to the official Python documentation.

7.3 Word counting

Assume that we have a file that contains a list of words, one word per line. We want to know how many times each word occurs in the file. Dictionaries can be used to do this easily:

```
>>> sample_string = "To be or not to be"
>>> occurrences = {}
>>> for word in sample_string.split():
...     occurrences[word] = occurrences.get(word, 0) + 1      ❶
...
>>> for word in occurrences:
...     print("The word", word, "occurs", occurrences[word], \
...             "times in the string")
...
```

We increment the `occurrences` count for each word ❶. This is a good example of the power of dictionaries. The code is not only simple, but because dictionary operations are highly optimized in Python, it's also quite fast.

7.4 What can be used as a key?

The previous examples use strings as keys, but Python permits more than just strings to be used in this manner. Any Python object that is immutable and hashable can be used as a key to a dictionary.

In Python, as discussed earlier, any object that can be modified is called *mutable*. Lists are mutable, because list elements can be added, changed, or removed. Dictionaries are also mutable, for the same reason. Numbers are immutable. If a variable `x` is

holding the number 3, and you assign 4 to `x`, you've changed the value in `x`, but you haven't changed the number 3; 3 is still 3. Strings are also immutable. `list[n]` returns the *n*th element of `list`, `string[n]` returns the *n*th character of `string`, and `list[n] = value` changes the *n*th element of `list`, but `string[n] = character` is illegal in Python and causes an error.

Unfortunately, the requirement that keys be immutable and hashable means that lists can't be used as dictionary keys. But there are many instances when it would be convenient to have a listlike key. For example, it's convenient to store information about a person under a key consisting of both their first and last names, which could be easily done if we could use a two-element list as a key.

Python solves this difficulty by providing tuples, which are basically immutable lists—they're created and used similarly to lists, except that when you have them, you can't modify them. But there's one further restriction: keys must also be hashable, which takes things a step further than just immutable. To be hashable, a value must have a hash value (provided by a `__hash__` method) that never changes throughout the life of the value. That means that tuples containing mutable values, although they themselves are immutable, aren't hashable. Only tuples that don't contain any mutable objects nested within them are hashable and valid to use as keys for dictionaries. Table 7.2 illustrates which of Python's built-in types are immutable, hashable, and eligible to be dictionary keys.

Table 7.2 Python values eligible to be used as dictionary keys

Python type	Immutable?	Hashable?	Dictionary key?
`int`	Yes	Yes	Yes
`float`	Yes	Yes	Yes
`boolean`	Yes	Yes	Yes
`complex`	Yes	Yes	Yes
`str`	Yes	Yes	Yes
`bytes`	Yes	Yes	Yes
`bytearray`	No	No	No
`list`	No	No	No
`tuple`	Yes	Sometimes	Sometimes
`set`	No	No	No
`frozenset`	Yes	Yes	Yes
`dictionary`	No	No	No

The next sections give examples illustrating how tuples and dictionaries can work together.

7.5 Sparse matrices

In mathematical terms, a *matrix* is a two-dimensional grid of numbers, usually written in textbooks as a grid with square brackets on each side, as shown at right.

```
3   0  -2  11
0   9   0   0
0   7   0   0
0   0   0  -5
```

A fairly standard way to represent such a matrix is by means of a list of lists. In Python, it's presented like this:

```
matrix = [[3, 0, -2, 11], [0, 9, 0, 0], [0, 7, 0, 0], [0, 0, 0, -5]]
```

Elements in the matrix can then be accessed by row and column number:

```
element = matrix[rownum][colnum]
```

But in some applications, such as weather forecasting, it's common for matrices to be very large—thousands of elements to a side, meaning millions of elements in total. It's also common for such matrices to contain many zero elements. In some applications, all but a small percentage of the matrix elements may be set to zero. In order to conserve memory, it's common for such matrices to be stored in a form where only the nonzero elements are actually stored. Such representations are called *sparse matrices*.

It's simple to implement sparse matrices using dictionaries with tuple indices. For example, the previous sparse matrix can be represented as follows:

```
matrix = {(0, 0): 3, (0, 2): -2, (0, 3): 11,
          (1, 1): 9, (2, 1): 7, (3, 3): -5}
```

Now, you can access an individual matrix element at a given row and column number by the following bit of code:

```
if (rownum, colnum) in matrix:
    element = matrix[(rownum, colnum)]
else:
    element = 0
```

A slightly less clear (but more efficient) way of doing this is to use the dictionary `get` method, which you can tell to return `0` if it can't find a key in the dictionary and otherwise return the value associated with that key. This avoids one of the dictionary lookups:

```
element = matrix.get((rownum, colnum), 0)
```

If you're considering doing extensive work with matrices, you may want to look into `NumPy`, the numeric computation package.

7.6 Dictionaries as caches

The following is an example of how dictionaries can be used as *caches*, data structures that store results to avoid recalculating those results over and over. A short while ago, I wrote a function called `sole`, which took three integers as arguments and returned a result. It looked something like this:

```
def sole(m, n, t):
    # . . . do some time-consuming calculations . . .
    return(result)
```

The problem with this function was that it really was time consuming, and because I was calling `sole` tens of thousands of times, the program ran too slowly.

But `sole` was called with only about 200 different combinations of arguments during any program run. That is, I might call `sole(12, 20, 6)` some 50 or more times during the execution of my program and similarly for many other combinations of arguments. By eliminating the recalculation of `sole` on identical arguments, I'd save a huge amount of time. I used a dictionary with tuples as keys, like so:

```
sole_cache = {}
def sole(m, n, t):
    if (m, n, t) in sole_cache:
        return sole_cache[(m, n, t)]
    else:
        # . . . do some time-consuming calculations . . .
        sole_cache[(m, n, t)] = result
        return result
```

The rewritten `sole` function uses a global variable to store previous results. The global variable is a dictionary, and the keys of the dictionary are tuples corresponding to argument combinations that have been given to `sole` in the past. Then, any time `sole` passes an argument combination for which a result has already been calculated, it returns that stored result, rather than recalculating it.

7.7 Efficiency of dictionaries

If you come from a traditional compiled-language background, you may hesitate to use dictionaries, worrying that they're less efficient than lists (arrays). The truth is that the Python dictionary implementation is quite fast. Many of the internal language features rely on dictionaries, and a lot of work has gone into making them efficient. Because all of Python's data structures are heavily optimized, you shouldn't spend much time worrying about which is faster or more efficient. If the problem can be solved more easily and cleanly by using a dictionary than by using a list, do it that way, and consider alternatives only if it's clear that dictionaries are causing an unacceptable slowdown.

7.8 Summary

Dictionaries are a basic and powerful Python data structure, used for many purposes even within Python itself. The ability to use any immutable object as a key to retrieve a corresponding value makes dictionaries able to handle collections of data with less code and more direct access than many other solutions.

We've now surveyed the main data structures in Python, so the next step is to look at the structures Python has to control the flow of a program.

8 Control flow

This chapter covers

- Repeating code with a `while` loop
- Making decisions: the `if-elif-else` statement
- Iterating over a list with a `for` loop
- Using list and dictionary comprehensions
- Delimiting statements and blocks with indentation
- Evaluating Boolean values and expressions

Python provides a complete set of control-flow elements, with loops and conditionals. This chapter examines each in detail.

8.1 The while loop

You've come across the basic `while` loop several times already. The full `while` loop looks like this:

```
while condition:
    body
else:
    post-code
```

condition is an expression that evaluates to a true or false value. As long as it's True, the body will be executed repeatedly. If it evaluates to False, the while loop will execute the post-code section and then terminate. If the condition starts out by being false, the body won't be executed at all—just the post-code section. The body and post-code are each sequences of one or more Python statements that are separated by newlines and are at the same level of indentation. The Python interpreter uses this level to delimit them. No other delimiters, such as braces or brackets, are necessary.

Note that the else part of the while loop is optional and not often used. That's because as long as there is no break in the body, this loop

```
while condition:
    body
else:
    post-code
```

and this loop

```
while condition:
    body
post-code
```

do the same things—and the second is simpler to understand. I probably wouldn't have mentioned the else clause except that if you haven't learned about it by now, you may have found it confusing if you found this syntax in another person's code. Also, it's useful in some situations.

8.1.1 The break and continue statements

The two special statements break and continue can be used in the body of a while loop. If break is executed, it immediately terminates the while loop, and not even the post-code (if there is an else clause) will be executed. If continue is executed, it causes the remainder of the body to be skipped over; the condition is evaluated again, and the loop proceeds as normal.

8.2 The if-elif-else statement

The most general form of the if-then-else construct in Python is

```
if condition1:
    body1
elif condition2:
    body2
elif condition3:
    body3
.
.
.
elif condition(n-1):
    body(n-1)
```

```
else:
    body(n)
```

It says: if `condition1` is `true`, execute `body1`; otherwise, if `condition2` is `true`, execute `body2`; otherwise … and so on, until it either finds a condition that evaluates to `True` or hits the `else` clause, in which case it executes `body(n)`. As for the `while` loop, the `body` sections are again sequences of one or more Python statements that are separated by newlines and are at the same level of indentation.

Of course, you don't need all that luggage for every conditional. You can leave out the `elif` parts, or the `else` part, or both. If a conditional can't find any body to execute (no conditions evaluate to `True`, and there is no `else` part), it does nothing.

The `body` after the `if` statement is required. But you can use the `pass` statement here (as you can anywhere in Python where a statement is required). The `pass` statement serves as a placeholder where a statement is needed, but it performs no action:

```
if x < 5:
    pass
else:
    x = 5
```

There is no case (or switch) statement in Python.

8.3 The for loop

A `for` loop in Python is different from `for` loops in some other languages. The traditional pattern is to increment and test a variable on each iteration, which is what C `for` loops usually do. In Python, a `for` loop iterates over the values returned by any iterable object—that is, any object that can yield a sequence of values. For example, a `for` loop can iterate over every element in a list, a tuple, or a string. But an iterable object can also be a special function called `range` or a special type of function called a *generator*. This can be quite powerful. The general form is

```
for item in sequence:
    body
else:
    post-code
```

`body` will be executed once for each element of `sequence`. `variable` is set to be the first element of `sequence`, and `body` is executed; then, `variable` is set to be the second element of `sequence`, and `body` is executed; and so on, for each remaining element of the `sequence`.

The `else` part is optional. As with the `else` part of a `while` loop, it's rarely used. `break` and `continue` do the same thing in a `for` loop as in a `while` loop.

This small loop prints out the reciprocal of each number in `x`:

```
x = [1.0, 2.0, 3.0]
for n in x:
    print(1 / n)
```

8.3.1 The range function

Sometimes you need to loop with explicit indices (to use the position at which values occur in a list). You can use the range command together with the len command on lists to generate a sequence of indices for use by the for loop. This code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i in range(len(x)):
    if x[i] < 0:
        print("Found a negative number at index ", i)
```

Given a number *n*, range(n) returns a sequence 0, 1, 2, …, *n*–2, *n*–1. So, passing it the length of a list (found using len) produces a sequence of the indices for that list's elements. The range function doesn't build a Python list of integers—it just appears to. Instead, it creates a range object that produces integers on demand. This is useful when you're using explicit loops to iterate over really large lists. Instead of building a list with 10 million elements in it, for example, which would take up quite a bit of memory, you can use range(10000000), which takes up only a small amount of memory and generates a sequence of integers from 0 up to 10000000 as needed by the for loop.

CONTROLLING STARTING AND STEPPING VALUES WITH RANGE

You can use two variants on the range function to gain more control over the sequence it produces. If you use range with two numeric arguments, the first argument is the starting number for the resulting sequence and the second number is the number the resulting sequence goes up to (but doesn't include). Here are a few examples:

```
>>> list(range(3, 7))          <— 1
[3, 4, 5, 6]
>>> list(range(2, 10))         <— 1
[2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 3))
[]
```

list() is used only to force the items range would generate to appear as a list. It's not normally used in actual code (1).

This still doesn't allow you to count backward, which is why the value of range(5, 3) is an empty list. To count backward, or to count by any amount other than 1, you need to use the optional third argument to range, which gives a step value by which counting proceeds:

```
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
>>> list(range(5, 0, -1))
[5, 4, 3, 2, 1]
```

Sequences returned by range always include the starting value given as an argument to range and never include the ending value given as an argument.

8.3.2 Using break and continue in for loops

The two special statements `break` and `continue` can also be used in the `body` of a `for` loop. If `break` is executed, it immediately terminates the `for` loop, and not even the `post-code` (if there is an `else` clause) will be executed. If `continue` is executed in a `for` loop, it causes the remainder of the `body` to be skipped over, and the loop proceeds as normal with the next item. .

8.3.3 The for loop and tuple unpacking

You can use tuple unpacking to make some `for` loops cleaner. The following code takes a list of two-element tuples and calculates the value of the sum of the products of the two numbers in each tuple (a moderately common mathematical operation in some fields):

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0
for t in somelist:
    result = result + (t[0] * t[1])
```

Here's the same thing, but cleaner:

```
somelist = [(1, 2), (3, 7), (9, 5)]
result = 0

for x, y in somelist:
    result = result + (x * y)
```

We use a tuple `x, y` immediately after the `for` keyword, instead of the usual single variable. On each iteration of the for loop, `x` contains element 0 of the current tuple from `list`, and `y` contains element 1 of the current tuple from `list`. Using a tuple in this manner is a convenience of Python, and doing this indicates to Python that each element of the list is expected to be a tuple of appropriate size to unpack into the variable names mentioned in the tuple after the `for`.

8.3.4 The enumerate function

You can combine tuple unpacking with the `enumerate` function to loop over both the items and their index. This is similar to using `range` but has the advantage that the code is clearer and easier to understand. Like the previous example, the following code prints out all the positions in a list where it finds negative numbers:

```
x = [1, 3, -7, 4, 9, -5, 4]
for i, n in enumerate(x):                                    ❶
    if n < 0:                                        ❷
        print("Found a negative number at index ", i)        ❸
```

The `enumerate` function returns tuples of (index, item) ❶. You can access the item without the index ❷. The index is also available ❸.

8.3.5 The zip function

Sometimes it's useful to combine two or more iterables before looping over them. The `zip` function will take the corresponding elements from one or more iterables and combine them into tuples until it reaches the end of the shortest iterable:

```
>>> x = [1, 2, 3, 4]
>>> y = ['a', 'b', 'c']          <- y is 3 elements; x is 4 elements
>>> z = zip(x, y)
>>> list(z)
[(1, 'a'), (2, 'b'), (3, 'c')]   <- z has only 3 elements
```

8.4 List and dictionary comprehensions

The pattern of using a `for` loop to iterate through a list, modify or select individual elements, and create a new list or dictionary is very common. Such loops often look a lot like the following:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = []
>>> for item in x:
...     x_squared.append(item * item)
...
>>> x_squared
[1, 4, 9, 16]
```

This sort of situation is so common that Python has a special shortcut for such operations, called a *comprehension*. You can think of a list or dictionary comprehension as a one-line `for` loop that creates a new list or dictionary from another list. The pattern of a list comprehension is as follows:

```
new_list = [expression for variable in old_list if expression]
```

And a dictionary comprehension looks like this:

```
new_dict = {expression:expression for variable in list if expression}
```

In both cases, the heart of the expression is similar to the beginning of a `for` loop—`for variable in list`—with some expression using that variable to create a new key or value and an optional conditional expression using the value of the variable to select whether it's included in the new list or dictionary. For example, the following code does exactly the same thing as the previous code but is a list comprehension:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x]
>>> x_squared
[1, 4, 9, 16]
```

You can even use `if` statements to select items from the list:

```
>>> x = [1, 2, 3, 4]
>>> x_squared = [item * item for item in x if item > 2]
```

```
>>> x_squared
[9, 16]
```

Dictionary comprehensions are similar, but you need to supply both a key and a value. If we want to do something similar to the previous example but have the number be the key and the number's square be the value in a dictionary, we can use a dictionary comprehension, like so:

```
>>> x = [1, 2, 3, 4]
>>> x_squared_dict = {item: item * item for item in x}
>>> x_squared_dict
{1: 1, 2: 4, 3: 9, 4: 16}
```

List and dictionary comprehensions are very flexible and powerful, and when you get used to them they make list-processing operations much simpler. I recommend that you experiment with them and try them anytime you find yourself writing a `for` loop to process a list of items.

8.5 Statements, blocks, and indentation

Because the control flow constructs we encountered in this chapter are the first to make use of blocks and indentation, this is a good time to revisit the subject.

Python uses the indentation of the statements to determine the delimitation of the different blocks (or bodies) of the control-flow constructs. A block consists of one or more statements, which are usually separated by newlines. Examples of Python statements are the assignment statement, function calls, the `print` function, the placeholder `pass` statement, and the `del` statement. The control-flow constructs (`if-elif-else`, `while`, and `for` loops) are compound statements:

```
compound statement clause:
    block
compound statement clause:
    block
```

A compound statement contains one or more clauses that are each followed by indented blocks. Compound statements can appear in blocks just like any other statements. When they do, they create nested blocks.

You may also encounter a couple of special cases. Multiple statements may be placed on the same line if they are separated by semicolons. A block containing a single line may be placed on the same line after the semicolon of a clause of a compound statement:

```
>>> x = 1; y = 0; z = 0
>>> if x > 0: y = 1; z = 10
... else: y = -1
...
>>> print(x, y, z)
1 1 10
```

Improperly indented code will result in an exception being raised. You may encounter two forms of this. The first is

```
>>>
>>>   x = 1
File "<stdin>", line 1
    x = 1
    ^
   IndentationError: unexpected indent
>>>
```

We indented a line that should not have been indented. In the basic mode, the carat (^) indicates the spot where the problem occurred. In the IDLE Python Shell (see figure 8.1), the invalid indent is highlighted. The same message would occur if we didn't indent where necessary (that is, the first line after a compound statement clause).

One situation where this can occur can be confusing. If you're using an editor that displays tabs in four-space increments (or the Windows interactive mode, which indents the first tab only four spaces from the prompt) and indent one line with four spaces and then the next line with a tab, the two lines may appear to be at the same level of indentation. But you'll receive this exception because Python maps the tab to eight spaces. The best way to avoid this problem is to use only spaces in Python code. If you must use tabs for indentation, or if you're dealing with code that uses tabs, be sure never to mix them with spaces.

On the subject of the basic interactive mode and the IDLE Python Shell, you likely have noticed that you need an extra line after the outermost level of indentation:

```
>>> x = 1
>>> if x == 1:
...     y = 2
...     if v > 0:
...         z = 2
...         v = 0
...
>>> x = 2
```

No line is necessary after the line `z = 2`, but one is needed after the line `v = 0`. This line is unnecessary if you're placing your code in a module in a file.

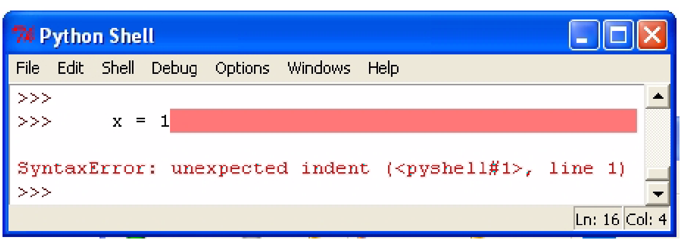


Figure 8.1 Indentation error

The second form of exception will occur if you indent a statement in a block less than the legal amount:

```
>>> x = 1
>>> if x == 1:
           y = 2
        z = 2
File "<stdin>", line 3
       z = 2
       ^
    IndentationError: unindent does not match any outer indentation level
```

Here, the line containing `z = 2` isn't lined up properly under the line containing `y = 2`. This form is rare, but I mention it again because in a similar situation, it may be confusing.

Python will allow you to indent any amount and won't complain regardless of how much you vary it as long as you're consistent within a single block. Please don't take improper advantage of this. The recommended standard is to use four spaces for each level of indentation.

Before leaving indentation, I'll cover breaking up statements across multiple lines. This of course is necessary more often as the level of indentation increases. You can explicitly break up a line using the backslash character. You can also implicitly break any statement between tokens when within a set of `()`, {}, or `[]` delimiters (that is, when typing a set of values in a list, a tuple, or a dictionary, or a set of arguments in a function call, or any expression within a set of brackets). You can indent the continuation line of a statement to any level you desire:

```
>>> print('string1', 'string2', 'string3' \
...    , 'string4', 'string5')
string1 string2 string3 string4 string5
>>> x = 100 + 200 + 300 \
...    + 400 + 500
>>> x
1500
>>> v = [100, 300, 500, 700, 900,
...    1100, 1300]
>>> v
[100, 300, 500, 700, 900, 1100, 1300]
>>> max(1000, 300, 500,
...        800, 1200)
1200
>>> x = (100 + 200 + 300
...          + 400 + 500)
>>> x
1500
```

You can break a string with a \ as well. But any indentation tabs or spaces will become part of the string, and the line *must* end with the \. To avoid this, you can use the fact that any set of string literals separated by whitespace is automatically concatenated:

```
>>> "strings separated by whitespace "    \
...    """are automatically"""  ' concatenated'
'strings separated by whitespace are automatically concatenated'
>>> x = 1
>>> if x > 0:
...        string1 = "this string broken by a backslash will end up \
...                with the indentation tabs in it"
...
>>> string1
'this string broken by a backslash will end up \t\t\twith
     the indentation tabs in it'
>>> if x > 0:
...        string1 = "this can be easily avoided by splitting the " \
...            "string in this way"
...
>>> string1
'this can be easily avoided by splitting the string in this way'
```

8.6 Boolean values and expressions

The previous examples of control flow use conditional tests in a fairly obvious manner but never really explain what constitutes true or false in Python or what expressions can be used where a conditional test is needed. This section describes these aspects of Python.

Python has a Boolean object type that can be set to either `True` or `False`. Any expression with a Boolean operation will return `True` or `False`.

8.6.1 Most Python objects can be used as Booleans

In addition, Python is similar to C with respect to Boolean values, in that C uses the integer 0 to mean false and any other integer to mean true. Python generalizes this idea; 0 or empty values are `False`, and any other values are `True`. In practical terms, this means the following:

- The numbers 0, 0.0, and 0+0j are all `False`; any other number is `True`.
- The empty string `""` is `False`; any other string is `True`.
- The empty list `[]` is `False`; any other list is `True`.
- The empty dictionary {} is `False`; any other dictionary is `True`.
- The empty set `set()` is `False`; any other set is `True`.
- The special Python value `None` is always `False`.

There are some Python data structures we haven't looked at yet, but generally the same rule applies. If the data structure is empty or 0, it's taken to mean false in a Boolean context; otherwise it's taken to mean true. Some objects, such as file objects and code objects, don't have a sensible definition of a 0 or empty element, and these objects shouldn't be used in a Boolean context.

8.6.2 Comparison and Boolean operators

You can compare objects using normal operators: `<`, `<=`, `>`, `>=`, and so forth. `==` is the equality test operator, and either `!=` or `<>` may be used as the "not equal to" test. There are also `in` and `not in` operators to test membership in sequences (lists, tuples, strings, and dictionaries) as well as `is` and `is not` operators to test whether two objects are the same.

Expressions that return a Boolean value may be combined into more complex expressions using the `and`, `or`, and `not` operators. This code snippet checks to see if a variable is within a certain range:

```
if 0 < x and x < 10:
    ...
```

Python offers a nice shorthand for this particular type of compound statement; you can write it as you would in a math paper:

```
if 0 < x < 10:
    ...
```

Various rules of precedence apply; when in doubt, you can use parentheses to make sure Python interprets an expression the way you want it to. This is probably a good idea for complex expressions, regardless of whether it's necessary, because it makes it clear to future maintainers of the code exactly what's happening. See the appendix for more details on precedence.

The rest of this section provides more advanced information. If this is your first read through this book as you're learning the language, you may want to skip over it.

The `and` and `or` operators return objects. The `and` operator returns either the first false object (that an expression evaluates to) or the last object. Similarly, the `or` operator returns either the first true object or the last object. As with many other languages, evaluation stops as soon as a true expression is found for the `or` operator or as soon as a false expression is found for the `and` operator:

```
>>> [2] and [3, 4]
[3, 4]
>>> [] and 5
[]
>>> [2] or [3, 4]
[2]
>>> [] or 5
5
>>>
```

The == and != operators test to see if their operands contains the same values. They are used in most situations. The is and is not operators test to see if their operands are the same object:

```
>>> x = [0]
>>> y = [x, 1]
>>> x is y[0]          <- They reference the same object
True
>>> x = [0]            <- x has been assigned to a different object
>>> x is y[0]
False
>>> x == y[0]
True
```

Revisit "Nested lists and deep copies" (section 5.6) of "Lists, tuples, and sets" (chapter 5) if this example isn't clear to you.

8.7 Writing a simple program to analyze a text file

To give you a better sense of how a Python program works, let's look a small sample that roughly replicates the UNIX wc utility and reports the number of lines, words, and characters in a file. The sample in listing 8.1 is deliberately written to be clear to programmers new to Python and as simple as possible.

Listing 8.1 word_count.py

```
#!/usr/bin/env python3.1

""" Reads a file and returns the number of lines, words,
    and characters - similar to the UNIX wc utility
"""

infile = open('word_count.tst')              <- Opens file

lines = infile.read().split("\n")            <- Reads file; splits into lines

line_count = len(lines)                      <- Gets number of lines with len()

word_count = 0                               | Initializes
char_count = 0                               | other counts

for line in lines:                           <- Iterates through lines

    words = line.split()                     <- Splits into words
    word_count += len(words)

    char_count += len(line)                  <- Returns number of characters

print("File has {0} lines, {1} words, {2} characters".format
                         count, word_count, char_count))       | Prints answers
```

To test, you can run this against a sample file containing the first paragraph of this chapter's summary, like listing 8.2.

Listing 8.2 word_count.tst

```
Python provides a complete set of control flow elements,
including while and for loops, and conditionals.
Python uses the level of indentation to group blocks
of code with control elements.
```

On running word_count.py, you'll get the following output:

```
vern@mac:~/quickpythonbook/code $ python3.1 word_count.py
File has 4 lines, 30 words, 189 characters
```

This code can give you an idea of a Python program. There isn't much code, and most of the work gets done in three lines of code in the `for` loop. Most Pythonistas see this conciseness as one of Python's great strengths.

8.8 Summary

Python provides a complete set of control-flow elements, including `while` and `for` loops and conditionals. Python uses the level of indentation to group blocks of code with control elements.

Python has the Boolean values `True` and `False`, which can be referenced by variables, but it also considers any 0 or empty value to be false and any nonzero or nonempty value to be true.

Control flow is an important part of programming, but just as important is the ability to package and reuse blocks of code. In the next few chapters, we'll look at ways to do that in Python, beginning with functions.

9 Functions

This chapter covers
- Defining functions
- Using function parameters
- Passing mutable objects as parameters
- Understanding local and global variables
- Creating and using lambda expressions
- Using decorators

This chapter assumes you're familiar with function definitions in at least one other computer language and with the concepts that correspond to function definitions, arguments, parameters, and so forth.

9.1 Basic function definitions

The basic syntax for a Python function definition is

```
def name(parameter1, parameter2, . . .):
    body
```

As it does with control structures, Python uses indentation to delimit the body of the function definition. The following simple example puts the factorial code from

a previous section into a function body, so we can call a `fact` function to obtain the factorial of a number:

```
>>> def fact(n):
...     """Return the factorial of the given number."""      ❶
...     r = 1
...     while n > 0:
...         r = r * n
...         n = n - 1
...     return r                                           ❷
...
```

The second line ❶ is an optional *documentation string* or *docstring*. You can obtain its value by printing `fact.__doc__`. The intention of docstrings is to describe the external behavior of a function and the parameters it takes, whereas comments should document internal information about how the code works. Docstrings are strings that immediately follow the first line of a function definition and are usually triple quoted to allow for multiline descriptions. Browsing tools are available that extract the first line of document strings. It's a standard practice for multiline documentation strings to give a synopsis of the function in the first line, follow this with a blank second line, and end with the rest of the information. This line shows the value after the return is sent back to the code calling the function ❷.

> **Procedure or function?**
>
> In some languages, a function that doesn't return a value is called a *procedure*. Although you can (and will) write functions that don't have a `return` statement, they aren't really procedures. All Python procedures are functions; if no explicit `return` is executed in the procedure body, then the special Python value `None` is returned, and if `return arg` is executed, then the value `arg` is immediately returned. Nothing else in the function body is executed once a `return` has been executed. Because Python doesn't have true procedures, we'll refer to both types as *functions*.

Although all Python functions return values, it's up to you whether a function's return value is used:

```
>>> fact(4)                                   ❶
24                                  ❷
>>> x = fact(4)             ❸
>>> x
24
>>>
```

The return value isn't associated with a variable ❶. The `fact` function's value is printed in the interpreter only ❷. The return value is associated with the variable x ❸.

9.2 Function parameter options

Most functions need parameters, and each language has its own specifications for how function parameters are defined. Python is flexible and provides three options for defining function parameters. These are outlined in this section.

9.2.1 Positional parameters

The simplest way to pass parameters to a function in Python is by position. In the first line of the function, you specify definition variable names for each parameter; when the function is called, the parameters used in the calling code are matched to the function's parameter variables based on their order. The following function computes `x` to the power of `y`:

```
>>> def power(x, y):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
>>> power(3, 3)
27
```

This method requires that the number of parameters used by the calling code exactly match the number of parameters in the function definition, or a `TypeError` exception will be raised:

```
>>> power(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() takes exactly 2 positional arguments (1 given)
>>>
```

Default values

Function parameters can have default values, which you declare by assigning a default value in the first line of the function definition, like so:

```
def fun(arg1, arg2=default2, arg3=default3, . . .)
```

Any number of parameters can be given default values. Parameters with default values must be defined as the last parameters in the parameter list. This is because Python, like most languages, pairs arguments with parameters on a positional basis. There must be enough arguments to a function that the last parameter in that function's parameter list that doesn't have a default value gets an argument. See the next section, "Passing arguments by parameter name," for a more flexible mechanism.

The following function also computes x to the power of y. But if y isn't given in a call to the function, the default value of 2 is used, and the function is just the square function:

```
>>> def power(x, y=2):
...     r = 1
...     while y > 0:
...         r = r * x
...         y = y - 1
...     return r
...
```

You can see the effect of the default argument in the following interactive session:

```
>>> power(3, 3)
27
>>> power(3)
9
```

9.2.2 Passing arguments by parameter name

You can also pass arguments into a function using the name of the corresponding function parameter, rather than its position. Continuing with the previous interactive example, we can type

```
>>> power(2, 3)
8
>>> power(3, 2)
9
>>> power(y=2, x=3)
9
```

Because the arguments to power in the final invocation of it are named, their order is irrelevant; the arguments are associated with the parameters of the same name in the definition of power, and we get back 3^2. This type of argument passing is called *keyword passing*.

Keyword passing, in combination with the default argument capability of Python functions, can be highly useful when you're defining functions with large numbers of possible arguments, most of which have common defaults. For example, consider a function that's intended to produce a list with information about files in the current directory and that uses Boolean arguments to indicate whether that list should include information such as file size, last modified date, and so forth, for each file. We can define such a function along these lines

```
def list_file_info(size=False, create_date=False, mod_date=False, ...):
    ...get file names...
    if size:
        # code to get file sizes goes here
        if create_date:
            # code to get create dates goes here
    .
    .
    .
    return fileinfostructure
```

and then call it from other code using keyword argument passing to indicate that we want only certain information (in this example, the file size and modification date but *not* the creation date):

```
fileinfo = list_file_info(size=True, mod_date=True)
```

This type of argument handling is particularly suited for functions with very complex behavior, and one place such functions occur is in graphical user interfaces. If you ever use the Tkinter package to build GUIs in Python, you'll find that the use of optional, keyword-named arguments like this is invaluable.

9.2.3 Variable numbers of arguments

Python functions can also be defined to handle variable numbers of arguments. You can do this two different ways. One way handles the relatively familiar case where you wish to collect an unknown number of arguments at the end of the argument list into a list. The other method can collect an arbitrary number of keyword-passed arguments, which have no correspondingly named parameter in the function parameter list, into a dictionary. These two mechanisms are discussed next.

DEALING WITH AN INDEFINITE NUMBER OF POSITIONAL ARGUMENTS

Prefixing the final parameter name of the function with a `*` causes all excess non-keyword arguments in a call of a function (that is, those positional arguments not assigned to another parameter) to be collected together and assigned as a tuple to the given parameter. Here's a simple way to implement a function to find the maximum in a list of numbers.

First, implement the function:

```
>>> def maximum(*numbers):
...     if len(numbers) == 0:
...         return None
...     else:
...         maxnum = numbers[0]
...         for n in numbers[1:]:
...             if n > maxnum:
...                 maxnum = n
...         return maxnum
...
```

Now, test out the behavior of the function:

```
>>> maximum(3, 2, 8)
8
>>> maximum(1, 5, 9, -2, 2)
9
```

DEALING WITH AN INDEFINITE NUMBER OF ARGUMENTS PASSED BY KEYWORD

An arbitrary number of keyword arguments can also be handled. If the final parameter in the parameter list is prefixed with `**`, it will collect all excess *keyword-passed* arguments into a dictionary. The index for each entry in the dictionary will be the keyword (parameter name) for the excess argument. The value of that entry is the argument

itself. An argument passed by keyword is excess in this context if the keyword by which it was passed doesn't match one of the parameter names of the function.

For example:

```
>>> def example_fun(x, y, **other):
...     print("x: {0}, y: {1}, keys in 'other': {2}".format(x,
...           y, list(other.keys())))
...     other_total = 0
...     for k in other.keys():
...         other_total = other_total + other[k]
...     print("The total of values in 'other' is {0}".format(other_total))
```

Trying out this function in an interactive session reveals that it can handle arguments passed in under the keywords `foo` and `bar`, even though these aren't parameter names in the function definition:

```
>>> example_fun(2, y="1", foo=3, bar=4)
x: 2, y: 1, keys in 'other': ['foo', 'bar']
The total of values in 'other' is 7
```

9.2.4 Mixing argument-passing techniques

It's possible to use all of the argument-passing features of Python functions at the same time, although it can be confusing if not done with care. Rules govern what you can do. See the documentation for the details.

9.3 Mutable objects as arguments

Arguments are passed in by object reference. The parameter becomes a new reference to the object. For immutable objects (such as tuples, strings, and numbers), what is done with a parameter has no effect outside the function. But if you pass in a mutable object (for example, a list, dictionary, or class instance), any change made to the object will change what the argument is referencing outside the function. Reassigning the parameter doesn't affect the argument, as shown in figures 9.1 and 9.2:

```
>>> def f(n, list1, list2):
...    list1.append(3)
...    list2 = [4, 5, 6]
...    n = n + 1
...
>>> x = 5
>>> y = [1, 2]
>>> z = [4, 5]
>>> f(x, y, z)
>>> x, y, z
(5, [1, 2, 3], [4, 5])
```

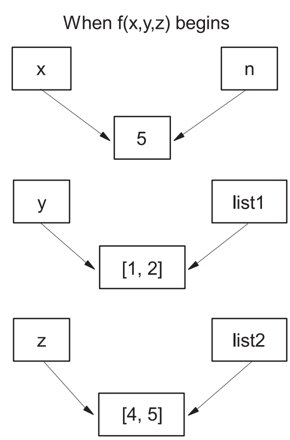


Figure 9.1 At the beginning of function `f()`, both the initial variables and the function parameters refer to the same objects.

Figures 9.1 and 9.2 illustrate what happens when function `f` is called. The variable `x` isn't changed because it's immutable. Instead, the function parameter `n` is set to refer to the new value of 6. Likewise, variable `z` is unchanged because inside function `f`, its corresponding parameter `list2` was set to refer to a new object, `[4, 5, 6]`. Only `y` sees a change because the actual list it points to was changed.

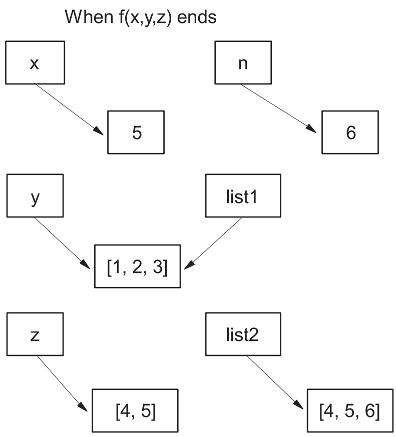


Figure 9.2 At the end of function `f()`, `y` (`list1` inside the function) has been changed internally, whereas `n` and `list2` refer to different objects.

9.4 Local, nonlocal, and global variables

Let's return to our definition of `fact` from the beginning of this chapter:

```
def fact(n):
    """Return the factorial of the given number."""
    r = 1
    while n > 0:
        r = r * n
        n = n - 1
    return r
```

Both the variables `r` and `n` are *local* to any particular call of the factorial function; changes to them made when the function is executing have no effect on any variables outside the function. Any variables in the parameter list of a function, and any variables created within a function by an assignment (like `r = 1` in `fact`), are local to the function.

You can explicitly make a variable global by declaring it so before the variable is used, using the `global` statement. Global variables can be accessed and changed by the function. They exist outside the function and can also be accessed and changed by other functions that declare them global or by code that's not within a function. Let's look at an example to see the difference between local and global variables:

```
>>> def fun():
...     global a
...     a = 1
...     b = 2
...
```

This defines a function that treats `a` as a global variable and `b` as a local variable and attempts to modify both `a` and `b`.

Now, test this function:

```
>>> a = "one"
>>> b = "two"
```

```
>>> fun()
>>> a
1
>>> b
'two'
```

The assignment to `a` within `fun` is an assignment to the global variable `a` also existing outside of `fun`. Because `a` is designated `global` in `fun`, the assignment modifies that global variable to hold the value `1` instead of the value `"one"`. The same isn't true for `b`—the local variable called `b` inside `fun` starts out referring to the same value as the variable `b` outside of `fun`, but the assignment causes `b` to point to a new value that's local to the function `fun`.

Similar to the `global` statement is the `nonlocal` statement, which causes an identifier to refer to a previously bound variable in the closest enclosing scope. We'll discuss scopes and namespaces in more detail in the next chapter, but the point is that `global` is used for a top-level variable, whereas `nonlocal` can refer to any variable in an enclosing scope, as the example in listing 9.1 illustrates.

Listing 9.1 File nonlocal.py

```
g_var = 0
nl_var = 0
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
def test():
    nl_var = 2
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
    def inner_test():
        global g_var
        nonlocal nl_var
        g_var = 1
        nl_var = 4
        print("in inner_test-> g_var: {0} nl_var: {1}".format(g_var,
                                                          nl_var))

    inner_test()
    print("in test-> g_var: {0} nl_var: {1}".format(g_var, nl_var))

test()
print("top level-> g_var: {0} nl_var: {1}".format(g_var, nl_var))
```

g_var in inner_test binds top-level g_var (annotation to `g_var = 0`)

nl_var in inner_test binds to nl_var in test (annotation to `nl_var = 2`)

g_var in inner_test binds top-level g_var (annotation to `global g_var`)

nl_var in inner_test binds to nl_var in test (annotation to `nonlocal nl_var`)

When run, this code prints the following:

```
top level-> g_var: 0 nl_var: 0
in test-> g_var: 0 nl_var: 2
in inner_test-> g_var: 1 nl_var: 4
in test-> g_var: 1 nl_var: 4
top level-> g_var: 1 nl_var: 0
```

Note that the value of the top-level `nl_var` hasn't been affected, which would happen if `inner_test` contained the line `global nl_var`.

The bottom line is that if you want to assign to a variable existing outside a function, you must explicitly declare that variable to be nonlocal or global. But if you're accessing a variable that exists outside the function, you don't need to declare it nonlocal or global. If Python can't find a variable name in the local function scope, it will attempt to look up the name in the global scope. Hence, accesses to global variables will automatically be sent through to the correct global variable. Personally, I don't recommend using this shortcut. It's much clearer to a reader if all global variables are explicitly declared as global. Further, you probably want to limit the use of global variables within functions to only rare occasions.

9.5 Assigning functions to variables

Functions can be assigned, like other Python objects, to variables, as shown in the following example:

```
>>> def f_to_kelvin(degrees_f):                         <-- Defines a function
...     return 273.15 + (degrees_f - 32) * 5 / 9
...
>>> def c_to_kelvin(degrees_c):                         <-- Defines a function
...     return 273.15 + degrees_c
...
>>> abs_temperature = f_to_kelvin                       <-- Assigns function to variable
>>> abs_temperature(32)
273.14999999999998
>>> abs_temperature = c_to_kelvin                       <-- Assigns function to variable
>>> abs_temperature(0)
273.14999999999998
```

You can place them in lists, tuples, or dictionaries:

```
>>> t = {'FtoK': f_to_kelvin, 'CtoK': c_to_kelvin}     <-- 1
>>> t['FtoK'](32)                                       <-- Accesses a function as value in dictionary
273.14999999999998
>>> t['CtoK'](0)                                        <-- Accesses a function as value in dictionary
273.14999999999998
```

A variable that refers to a function can be used in exactly the same way as the function **1**. This last example shows how you can use a dictionary to call different functions by the value of the strings used as keys. This is a common pattern in situations where different functions need to be selected based on a string value, and in many cases it takes the place of the `switch` structure found in languages like C and Java.

9.6 lambda expressions

Short functions like those you just saw can also be defined using `lambda` expressions of the form

```
lambda parameter1, parameter2, . . .: expression
```

`lambda` expressions are anonymous little functions that you can quickly define inline. Often, a small function needs to be passed to another function, like the key function

used by a list's sort method. In such cases, a large function is usually unnecessary, and it would be awkward to have to define the function in a separate place from where it's used. Our dictionary in the previous subsection can be defined all in one place with

```
>>> t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,     ❶
...       'CtoK': lambda deg_c: 273.15 + deg_c}
>>> t2['FtoK'](32)
273.14999999999998
```

This defines `lambda` expressions as values of the dictionary ❶. Note that `lambda` expressions don't have a `return` statement, because the value of the expression is automatically returned.

9.7 Generator functions

A *generator* function is a special kind of function that you can use to define your own iterators. When you define a generator function, you return each iteration's value using the `yield` keyword. When there are no more iterations, an empty `return` statement or flowing off the end of the function ends the iterations. Local variables in a generator function are saved from one call to the next, unlike in normal functions:

```
>>> def four():
...     x = 0                              <-- Sets initial value of x to 0
...     while x < 4:
...         print("in generator, x =", x)
...         yield x                        <-- Returns current value of x
...         x += 1                         <-- Increments value of x
...
>>> for i in four():
...     print(i)
...
in generator, x = 0
0
in generator, x = 1
1
in generator, x = 2
2
in generator, x = 3
3
```

Note that this generator function has a `while` loop that limits the number of times the generator will execute. Depending on how it's used, a generator that doesn't have some condition to halt it could cause an endless loop when called.

You can also use generator functions with `in` to see if a value is in the series that the generator produces:

```
>>> 2 in four()
in generator, x = 0
in generator, x = 1
in generator, x = 2
True
>>> 5 in four()
```

```
in generator, x = 0
in generator, x = 1
in generator, x = 2
in generator, x = 3
False
```

9.8 Decorators

Because functions are first-class objects in Python, they can be assigned to variables, as you've seen. Functions can be passed as arguments to other functions and passed back as return values from other functions.

For example, it's possible to write a Python function that takes another function as its parameter, wrap it in another function that does something related, and then return the new function. This new combination can be used instead of the original function:

```
>>> def decorate(func):
...     print("in decorate function, decorating", func.__name__)
...     def wrapper_func(*args):
...         print("Executing", func.__name__)
...         return func(*args)
...     return wrapper_func
...
>>> def myfunction(parameter):
...     print(parameter)
...
>>> myfunction = decorate(myfunction)
in decorate function, decorating myfunction
>>> myfunction("hello")
Executing myfunction
hello
```

A decorator is syntactic sugar for this process and lets you wrap one function inside another with a one-line addition. This still gives you exactly the same effect as the previous code, but the resulting code is much cleaner and easier to read.

Very simply, using a decorator involves two parts: defining the function that will be wrapping or "decorating" other functions and then using an `@` followed by the decorator immediately before the wrapped function is defined. The decorator function should take a function as a parameter and return a function, as follows:

```
>>> def decorate(func):
...     print("in decorate function, decorating", func.__name__)     ❶
...     def wrapper_func(*args):
...         print("Executing", func.__name__)
...         return func(*args)
...     return wrapper_func                                          ❷
...
>>> @decorate                                                        ❸
... def myfunction(parameter):
...     print(parameter)
...
in decorate function, decorating myfunction                          ❶
>>> myfunction("hello")                                              ❹
Executing myfunction
hello
```

The `decorate` function prints the name of the function it's wrapping when the function is defined ❶. When it's finished, the decorator returns the wrapped function ❷. `myfunction` is decorated using `@decorate` ❸. The wrapped function is called after the decorator function has completed ❹.

Using a decorator to wrap one function in another can be handy for a number of purposes. In web frameworks such as Django, decorators are used to make sure a user is logged in before executing a function; and in graphics libraries, decorators can be used to register a function with the graphics framework.

9.9 Summary

Defining functions in Python is simple but highly flexible. Although all variables created during the execution of a function body are local to that function, external variables can easily be accessed using the `global` statement.

Python functions provide exceedingly powerful argument-passing features:

- Arguments may be passed by position or by parameter name.
- Default values may be provided for function parameters.
- Functions can collect arguments into tuples, giving you the ability to define functions that take an indefinite number of arguments.
- Functions can collect arguments into dictionaries, giving you the ability to define functions that take an indefinite number of arguments passed by parameter name.

Functions are first-class objects in Python, which means that they can be assigned to variables, accessed by way of variables, and decorated. Functions are essential building blocks for writing readable, structured code. By packaging code that performs a particular function, they make reusing that code easier, and they also make the rest of your code simpler and easier to understand. The next step along this path is packaging functions (and other objects) into `modules`, which is the topic of the next chapter.

10 Modules and scoping rules

This chapter covers:

- Defining a module
- Writing a first module
- Using the `import` statement
- Modifying the module search path
- Making names private in modules
- Importing standard library and third-party modules
- Understanding Python scoping rules and namespaces

Modules are used to organize larger Python projects. The Python standard library is split into modules to make it more manageable. You don't need to organize your own code into modules, but if you're writing any programs that are more than a few pages long, or any code that you want to reuse, you should probably do so.

10.1 What is a module?

A *module* is a file containing code. A module defines a group of Python functions or other objects, and the name of the module is derived from the name of the file.

Modules most often contain Python source code, but they can also be compiled C or C++ object files. Compiled modules and Python source modules are used the same way.

As well as grouping related Python objects, modules help avoid name-clash problems. For example, you might write a module for your program called `mymodule`, which defines a function called `reverse`. In the same program, you might also wish to use somebody else's module called `othermodule`, which also defines a function called `reverse`, but which does something different from your `reverse` function. In a language without modules, it would be impossible to use two different functions named `reverse`. In Python, it's trivial—you refer to them in your main program as `mymodule.reverse` and `othermodule.reverse`.

This is because Python uses *namespaces*. A namespace is essentially a dictionary of the identifiers available to a block, function, class, module, and so on. We'll discuss namespaces a bit more at the end of this chapter, but be aware that each module has its own namespace, and this helps avoid naming conflicts.

Modules are also used to make Python itself more manageable. Most standard Python functions aren't built into the core of the language but instead are provided via specific modules, which you can load as needed.

10.2 A first module

The best way to learn about modules is probably to make one, so let's get started.

Create a text file called mymath.py, and in that text file enter the Python code in listing 10.1. (If you're using IDLE, select New Window from the File menu and start typing, as shown in figure 10.1.)

Listing 10.1 File mymath.py

```
"""mymath - our example math module"""
pi = 3.14159
def area(r):
    """area(r): return the area of a circle with radius r."""
    global pi
    return(pi * r * r)
```

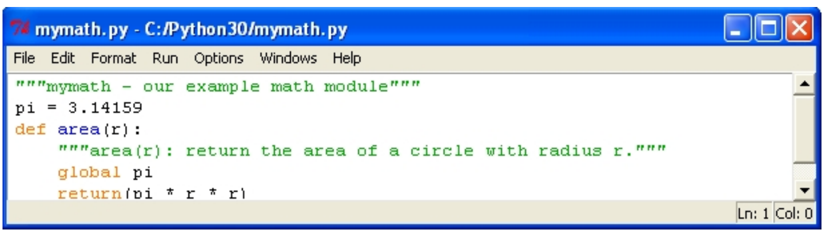


Figure 10.1 An IDLE edit window provides the same editing functionality as the shell window, including automatic indentation and colorization.

Save this for now in the directory where your Python executable is. This code merely assigns `pi` a value and defines a function. The .py filename suffix is strongly suggested for all Python code files. It identifies that file to the Python interpreter as consisting of Python source code. As with functions, you have the option of putting in a document string as the first line of your module.

Now, start up the Python Shell and type the following:

```
>>> pi
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> area(2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

In other words, Python doesn't have the constant `pi` or the function `area` built in.

Now, type

```
>>> import mymath
>>> pi
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'pi' is not defined
>>> mymath.pi
3.1415899999999999
>>> mymath.area(2)
12.56636
>>> mymath.__doc__
'mymath - our example math module'
>>> mymath.area.__doc__
'area(r): return the area of a circle with radius r.'
```

We've brought in the definitions for `pi` and `area` from the mymath.py file, using the `import` statement (which automatically adds on the .py suffix when it searches for the file defining the module named `mymath`). But the new definitions aren't directly accessible; typing `pi` by itself gave an error, and typing `area(2)` by itself would give an error. Instead, we access `pi` and `area` by *prepending* them with the name of the module that contains them. This guarantees name safety. There may be another module out there that also defines `pi` (maybe the author of that module thinks that pi is 3.14 or 3.14159265), but that is of no concern. Even if that other module is imported, its version of `pi` will be accessed by `othermodulename.pi`, which is different from `mymath.pi`. This form of access is often referred to as *qualification* (that is, the variable `pi` is being qualified by the module `mymath`). We may also refer to `pi` as an *attribute* of `mymath`.

Definitions within a module can access other definitions within that module, without prepending the module name. The `mymath.area` function accesses the `mymath.pi` constant as just `pi`.

If you want to, you can also specifically ask for names from a module to be imported in such a manner that you don't have to prepend it with the module name. Type

```
>>> from mymath import pi
>>> pi
3.1415899999999999
>>> area(2)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name 'area' is not defined
```

The name `pi` is now directly accessible because we specifically requested it using `from module import name`.

The function `area` still needs to be called as `mymath.area`, though, because it wasn't explicitly imported.

You may want to use the basic interactive mode or IDLE's Python shell to incrementally test a module as you're creating it. But if you change your module on disk, retyping the `import` command won't cause it to load again. You need to use the `reload` function from the `imp` module for this. The `imp` module provides an interface to the mechanisms behind importing modules:

```
>>> import mymath, imp
>>> imp.reload(mymath)
<module 'mymath' from '/home/doc/quickpythonbook/code/mymath.py'>
```

When a module is reloaded (or imported for the first time), all of its code is parsed. A syntax exception is raised if an error is found. On the other hand, if everything is okay, a .pyc file (for example, mymath.pyc) containing Python byte code is created.

Reloading a module doesn't put you back into exactly the same situation as when you start a new session and import it for the first time. But the differences won't normally cause you any problems. If you're interested, you can look up `reload` in the section on the `imp` module in the *Python Language Reference* to find the details.

Of course, modules don't need to be used from the interactive Python shell. You can also import them into scripts, or other modules for that matter; enter suitable `import` statements at the beginning of your program file. Internally to Python, the interactive session and a script are considered modules as well.

To summarize:

- A module is a file defining Python objects.
- If the name of the module file is modulename.py, then the Python name of the module is `modulename`.
- You can bring a module named `modulename` into use with the `import modulename` statement. After this statement is executed, objects defined in the module can be accessed as `modulename.objectname`.
- Specific names from a module can be brought directly into your program using the `from modulename import objectname` statement. This makes `objectname` accessible to your program without needing to prepend it with `modulename`, and it's useful for bringing in names that are often used.

10.3 The import statement

The `import` statement takes three different forms. The most basic,

```
import modulename
```

searches for a Python module of the given name, parses its contents, and makes it available. The importing code can use the contents of the module, but any references by that code to names within the module must still be prepended with the module name. If the named module isn't found, an error will be generated. Exactly where Python looks for modules will be discussed shortly.

The second form permits specific names from a module to be explicitly imported into the code:

```
from modulename import name1, name2, name3, . . .
```

Each of `name1`, `name2`, and so forth from within `modulename` is made available to the importing code; code after the `import` statement can use any of `name1`, `name2`, `name3`, and so on without prepending the module name.

Finally, there's a general form of the `from . . . import . . .` statement:

```
from modulename import *
```

The `*` stands for all the exported names in `modulename`. This imports all public names from `modulename`—that is, those that don't begin with an underscore, and makes them available to the importing code without the necessity of prepending the module name. But if a list of names called `__all__` exists in the module (or the package's __init__.py), then the names are the ones imported, whether they begin with an underscore or not.

You should take care when using this particular form of importing. If two modules both define a name, and you import both modules using this form of importing, you'll end up with a name clash, and the name from the second module will replace the name from the first. It also makes it more difficult for readers of your code to determine where names you're using originate. When you use either of the two previous forms of the import statement, you give your reader explicit information about where they're from.

But some modules (such as `tkinter`, which will be covered later) name their functions to make it obvious where they originate and to make it unlikely that name clashes will occur. It's also common to use the general import to save keystrokes when using an interactive shell.

10.4 The module search path

Exactly where Python looks for modules is defined in a variable called `path`, which you can access through a module called `sys`. Enter the following:

```
>>> import sys
>>> sys.path
_list of directories in the search path_
```

The value shown in place of `_list of directories in the search path_` will depend on the configuration of your system. Regardless of the details, the string indicates a list of directories that Python searches (in order) when attempting to execute an `import` statement. The first module found that satisfies the `import` request is used. If there's no satisfactory module in the module search path, an `ImportError` exception is raised.

If you're using IDLE, you can graphically look at the search path and the modules on it using the Path Browser window, which you can start from File menu of the Python Shell window.

The `sys.path` variable is initialized from the value of the environment (operating system) variable `PYTHONPATH`, if it exists, or from a default value that's dependent on your installation. In addition, whenever you run a Python script, the `sys.path` variable for that script has the directory containing the script inserted as its first element—this provides a convenient way of determining where the executing Python program is located. In an interactive session such as the previous one, the first element of `sys.path` is set to the empty string, which Python takes as meaning that it should first look for modules in the current directory.

10.4.1 Where to place your own modules

In the example that started this chapter, the `mymath` module was accessible to Python because (1) when you execute Python interactively, the first element of `sys.path` is `""`, telling Python to look for modules in the current directory; and (2) you were executing Python in the directory that contained the mymath.py file. In a production environment, neither of these conditions will typically be true. You won't be running Python interactively, and Python code files won't be located in your current directory. In order to ensure that your programs can use modules you coded, you need to do one of the following:

- Place your modules into one of the directories that Python normally searches for modules.
- Place all the modules used by a Python program into the same directory as the program.
- Create a directory (or directories) that will hold your modules, and modify the `sys.path` variable so that it includes this new directory.

Of these three options, the first is apparently the easiest and is also an option that you should *never* choose unless your version of Python includes local code directories in its default module search path. Such directories are specifically intended for site-specific code and aren't in danger of being overwritten by a new Python install because they're not part of the Python installation. If your `sys.path` refers to such directories, you can put your modules there.

The second option is a good choice for modules that are associated with a particular program. Just keep them with the program.

The third option is the right choice for site-specific modules that will be used in more than one program at that site. You can modify `sys.path` in various ways. You can

assign to it in your code, which is easy, but doing so hard-codes directory locations into your program code; you can set the `PYTHONPATH` environment variable, which is relatively easy, but it may not apply to all users at your site; or you can add to the default search path using a .pth file.

See the section on environment variables in the appendix for examples of how to set `PYTHONPATH`. The directory or directories you set it to are prepended to the `sys.path` variable. If you use it, be careful that you don't define a module with the same name as one of the existing library modules that you're using or is being used for you. Your module will be found before the library module. In some cases, this may be what you want, but probably not often.

You can avoid this issue using the .pth method. In this case, the directory or directories you added will be appended to `sys.path`. The last of these mechanisms is best illustrated by a quick example. On Windows, you can place this in the directory pointed to by `sys.prefix`. Assume your `sys.prefix` is `c:\program files\python`, and place the file in listing 10.2 in that directory.

Listing 10.2 File myModules.pth

```
mymodules
c:\My Documents\python\modules
```

The next time a Python interpreter is started, `sys.path` will have `c:\program files\python\mymodules` and `c:\My Documents\python\modules` added to it, if they exist. You can now place your modules in these directories. Note that the mymodules directory still runs the danger of being overwritten with a new installation. The modules directory is safer. You also may have to move or create a mymodules.pth file when you upgrade Python. See the description of the `site` module in the *Python Library Reference* if you want more details on using .pth files.

10.5 Private names in modules

We mentioned that you can enter `from module import *` to import *almost* all names from a module. The exception to this is that names in the module beginning with an underscore can't be imported in this manner so that people can write modules that are intended for importation with `from module import *`. By starting all internal names (that is, names that shouldn't be accessed outside the module) with an underscore, you can ensure that `from module import *` brings in only those names that the user will want to access.

To see this in action, let's assume we have a file called modtest.py, containing the code in listing 10.3.

Listing 10.3 File modtest.py

```
"""modtest: our test module"""
def f(x):
    return x
```

```
def _g(x):
    return x
a = 4
_b = 2
```

Now, start up an interactive session, and enter the following:

```
>>> from modtest import *
>>> f(3)
3
>>> _g(3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name '_g' is not defined
>>> a
4
>>> _b
Traceback (innermost last):
  File "<stdin>", line 1, in ?
NameError: name '_b' is not defined
```

As you can see, the names `f` and `a` are imported, but the names `_g` and `_b` remain hidden outside of `modtest`. Note that this behavior occurs only with `from ... import *`. We can do the following to access `_g` or `_b`:

```
>>> import modtest
>>> modtest._b
2
>>> from modtest import _g
>>> _g(5)
5
```

The convention of leading underscores to indicate private names is used throughout Python and not just in modules. You'll encounter it in classes and packages, later in the book.

10.6 Library and third-party modules

At the beginning of this chapter, I mentioned that the standard Python distribution is split into modules to make it more manageable. After you've installed Python, all the functionality in these library modules is available to you. All that's needed is to import the appropriate modules, functions, classes, and so forth explicitly, before you use them.

Many of the most common and useful standard modules are discussed throughout this book. But the standard Python distribution includes far more than what this book describes. At the very least, you should browse through the table of contents of the *Python Library Reference.*

In IDLE, you can easily browse to and look at those written in Python using the Path Browser window. You can also search for example code that uses them with the Find in Files dialog box, which you can open from the Edit menu of the Python Shell window. You can search through your own modules as well in this way.

Available third-party modules, and links to them, are identified on the Python home page. You need to download these and place them in a directory in your module search path in order to make them available for import into your programs.

10.7 Python scoping rules and namespaces

Python's scoping rules and namespaces will become more interesting as your experience as a Python programmer grows. If you're new to Python, you probably don't need to do anything more than quickly read through the text to get the basic ideas. For more details, look up "namespaces" in the *Python Language Reference*.

The core concept here is that of a *namespace*. A namespace in Python is a mapping from identifiers to objects and is usually represented as a dictionary. When a block of code is executed in Python, it has three namespaces: *local*, *global*, and *built-in* (see figure 10.2).

When an identifier is encountered during execution, Python first looks in the *local namespace* for it. If it isn't found, the *global namespace* is looked in next. If it still hasn't been found, the *built-in namespace* is checked. If it doesn't exist there, this is considered an error and a `NameError` exception occurs.

For a module, a command executed in an interactive session, or a script running from a file, the global and local namespaces are the same. Creating any variable or function or importing anything from another module results in a new entry, or *binding*, being made in this namespace.

But when a function call is made, a local namespace is created, and a binding is entered in it for each parameter of the call. A new binding is then entered into this local namespace whenever a variable is created within the function. The global namespace of a function is the global namespace of the containing block of the function (that of the module, script file, or interactive session). It's independent of the dynamic context from which it's called.

In all of these situations, the built-in namespace is that of the `__builtins__` module. This module contains, among other things, all the built-in functions you've encountered (such as `len`, `min`, `max`, `int`, `float`, `long`, `list`, `tuple`, `cmp`, `range`, `str`, and `repr`) and the other built-in classes in Python, such as the exceptions (like `NameError`).

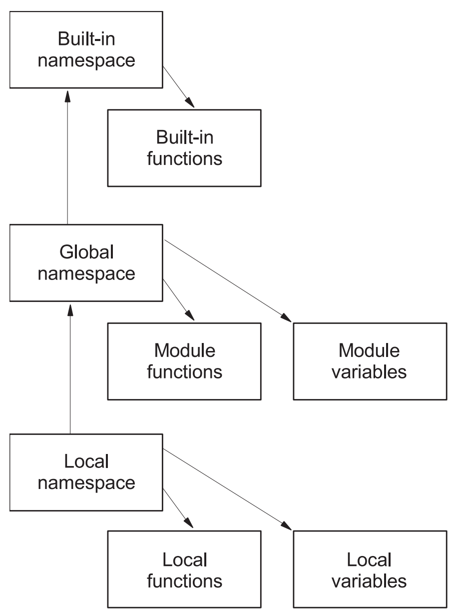


Figure 10.2 The order in which namespaces are checked to locate identifiers

One thing that sometimes catches new Python programmers is the fact that you can override items in the built-in module. If, for example, you create a list in your program and put it in a variable called `list`, you can't subsequently use the built-in `list` function. The entry for your list is found first. There's no differentiation between names for functions and modules and other objects. The most recent occurrence of a binding for a given identifier is used.

Enough talk—it's time to explore this with some examples. We use two built-in functions, `locals` and `globals`. They return dictionaries containing the bindings in the local and global namespaces, respectively.

Start a new interactive session:

```
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
➥'__doc__': None, '__package__': None}
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
➥'__doc__': None, '__package__': None}>>>
```

The local and global namespaces for this new interactive session are the same. They have three initial key/value pairs that are for internal use: (1) an empty documentation string `__doc__`, (2) the main module name `__name__` (which for interactive sessions and scripts run from files is always `__main__`), and (3) the module used for the built-in namespace `__builtins__` (the module `__builtins__`).

Now, if we continue by creating a variable and importing from modules, we'll see a number of bindings created:

```
>>> z = 2
>>> import math
>>> from cmath import cos
>>> globals()
{'cos': <built-in function cos>, '__builtins__': <module 'builtins'
➥(built-in)>, '__package__': None, '__name__': '__main__', 'z': 2,
➥'__doc__': None, 'math': <module 'math' from
➥'/usr/local/lib/python3.0/libdynload/math.so'>}
>>> locals()
{'cos': <built-in function cos>, '__builtins__':
➥ <module 'builtins' (built-in)>, '__package__': None, '__name__':
➥'__main__', 'z': 2, '__doc__': None, 'math': <module 'math' from
➥'/usr/local/lib/python3.0/libdynload/math.so'>}
>>> math.ceil(3.4)
4
```

As expected, the local and global namespaces continue to be equivalent. Entries have been added for `z` as a number, `math` as a module, and `cos` from the `cmath` module as a function.

You can use the `del` statement to remove these new bindings from the namespace (including the module bindings created with the `import` statements):

```
>>> del z, math, cos
>>> locals()
```

```
{'__builtins__': <module 'builtins' (built-in)>, '__package__': None,
'__name__': '__main__', '__doc__': None}
>>> math.ceil(3.4)
Traceback (innermost last):
  File "<stdin>", line 1, in <module>
NameError: math is not defined
>>> import math
>>> math.ceil(3.4)
4
```

The result isn't drastic, because we're able to import the `math` module and use it again. Using `del` in this manner can be handy when you're in the interactive mode.[1]

For the trigger happy, yes, it's also possible to use `del` to remove the `__doc__`, `__main__`, and `__builtins__` entries. But resist doing this, because it wouldn't be good for the health of your session!

Now, let's look at a function created in an interactive session:

```
>>> def f(x):
...     print("global: ", globals())
...     print("Entry local: ", locals())
...     y = x
...     print("Exit local: ", locals())
...
>>> z = 2
>>> globals()
{'f': <function f at 0xb7cbfeac>, '__builtins__': <module 'builtins'
➥ (built-in)>, '__package__': None, '__name__': '__main__', 'z': 2,
➥ '__doc__': None}
>>> f(z)
global:  {'f': <function f at 0xb7cbfeac>, '__builtins__': <module
➥ 'builtins' (built-in)>, '__package__': None, '__name__': '__main__',
➥ 'z': 2, '__doc__': None}
Entry local:  {'x': 2}
Exit local:  {'y': 2, 'x': 2}
>>>
```

If we dissect this apparent mess, we see that, as expected, upon entry the parameter `x` is the original entry in `f`'s local namespace, but `y` is added later. The global namespace is the same as that of our interactive session, because this is where `f` was defined. Note that it contains `z`, which was defined after `f`.

In a production environment, you normally call functions that are defined in modules. Their global namespace is that of the module they're defined in. Assume that we've created the file in listing 10.4.

Listing 10.4 File scopetest.py

```
"""scopetest: our scope test module"""
v = 6
```

[1] Using `del` and then `import` again won't pick up changes made to a module on disk. It isn't removed from memory and then loaded from disk again. The binding is taken out of and then put back into your namespace. You still need to use `imp.reload` if you want to pick up changes made to a file.

```
def f(x):
    """f: scope test function"""
    print("global: ", list(globals().keys()))
    print("entry local:", locals())
    y = x
    w = v
    print("exit local:", list(locals().keys()))
```

Note that we'll be printing only the keys (identifiers) of the dictionary returned by `globals`. This will reduce the clutter in the results. It was necessary in this case due to the fact that in modules as an optimization, the whole `__builtins__` dictionary is stored in the value field for the `__builtins__` key:

```
>>> import scopetest
>>> z = 2
>>> scopetest.f(z)
global:  ['f', '__builtins__', '__file__', '__package__', 'v', '__name__',
➥ '__doc__']
entry local: {'x': 2}
exit local: {'y': 2, 'x': 2, 'w': 6}
```

The global namespace is now that of the `scopetest` module and includes the function `f` and integer `v` (but not `z` from our interactive session). Thus, when creating a module, you have complete control over the namespaces of its functions.

We've now covered local and global namespaces. Next, let's move on to the built-in namespace. We'll introduce another built-in function, `dir`, which, given a module, returns a list of the names defined in it:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
     'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
     'Ellipsis', 'EnvironmentError', 'Exception', 'False',
     'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
     'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
     'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError',
     'NameError', 'None', 'NotImplemented', 'NotImplementedError', 'OSError',
     'OverflowError', 'PendingDeprecationWarning', 'ReferenceError',
     'RuntimeError', 'RuntimeWarning', 'StopIteration', 'SyntaxError',
     'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
     'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
     'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
     'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
     'ZeroDivisionError', '__build_class__', '__debug__', '__doc__',
     '__import__', '__name__', '__package__', 'abs', 'all', 'any', 'ascii',
     'bin', 'bool', 'bytearray', 'bytes', 'chr', 'classmethod', 'cmp',
     'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir',
     'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
     'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help',
     'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
     'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
     'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
     'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
     'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
     'vars', 'zip']
```

There are a lot of entries here. Those ending in `Error` and `System Exit` are the names of the exceptions built into Python. These will be discussed in chapter 14, "Exceptions."

The last group (from `abs` to `zip`) are built-in functions of Python. You've already seen many of these in this book and will see more. But they won't all be covered here. If you're interested, you can find details on the rest in the *Python Library Reference*. You can also at any time easily obtain the documentation string for any of them, either by using the `help()` function or by printing the docstring directly:

```
>>> print(max.__doc__)
max(iterable[, key=func]) -> value
max(a, b, c, ...[, key=func]) -> value

With a single iterable argument, return its largest item.
With two or more arguments, return the largest argument.
>>>
```

As mentioned earlier, it's not unheard of for a new Python programmer to inadvertently override a built-in function:

```
>>> list("Peyto Lake")
['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']
>>> list = [1, 3, 5, 7]
>>> list("Peyto Lake")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: 'list' object is not callable
```

The Python interpreter won't look beyond the new binding for `list` as a `list`, even though we're using the built-in `list` function syntax.

The same thing will, of course, happen if we try to use the same identifier twice in a single namespace. The previous value will be overwritten, regardless of its type:

```
>>> import mymath
>>> mymath = mymath.area
>>> mymath.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'function' object has no attribute 'pi'
```

When you're aware of this, it isn't a significant issue. Reusing identifiers, even for different types of objects, wouldn't make for the most readable code anyway. If you do inadvertently make one of these mistakes when in interactive mode, it's easy to recover. You can use `del` to remove your binding, to regain access to an overridden built-in, or to import your module again to regain access:

```
>>> del list
>>> list("Peyto Lake")
['P', 'e', 'y', 't', 'o', ' ', 'L', 'a', 'k', 'e']
>>> import mymath
>>> mymath.pi
3.1415899999999999
```

The locals and globals functions can be useful as simple debugging tools. The dir function doesn't give the current settings; but if you call it without parameters, it returns a sorted list of the identifiers in the local namespace. This helps catch the mistyped variable error that compilers may usually catch for you in languages that require declarations:

```
>>> x1 = 6
>>> xl = x1 - 2
>>> x1
6
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'x1', 'xl']
```

The debugger that's bundled with IDLE has settings where you can view the local and global variable settings as you step through your code; it displays the output of the locals and globals functions.

10.8 Summary

Python uses modules to manage Python code and objects but allows you to put related code and objects into a file. Not only does this make managing and reusing larger amounts of code easier, but it also helps avoid conflicting variable names, because each object imported from a module is normally named in association with its module.

Being able to package related functions into modules is the final piece of knowledge you need to write standalone programs and scripts in Python, and that's what we'll discuss in the next chapter.

11 Python programs

This chapter covers

- Creating a very basic program
- Making a program directly executable on Linux/UNIX
- Writing programs on Mac OS X
- Selecting execution options in Windows
- Comparing programs on Windows versus Linux/UNIX
- Combining programs and modules
- Distributing Python applications

Up until now, you've been using the Python interpreter mainly in interactive mode. For production use, you'll want to create Python programs or scripts. A number of the sections in this chapter focus on command-line programs. If you come from a Linux/UNIX background, you may be familiar with scripts that can be started from a command line and given arguments and options that can be used to pass in information and possibly redirect their input and output. If you're from a Windows or Mac background, these things may be new to you, and you may be more inclined to question their value.

It's true that command-line scripts are less convenient to use in a GUI environment, but the Mac now has the option of a UNIX command-line shell, and Windows

also offers enhanced command-line options. It will be well worth your time to read the bulk of this chapter at some point. You may find occasions when these techniques are useful, or you may run across code that you need to understand that uses some of them. In particular, command-line techniques are very useful when you need to process large numbers of files.

11.1 Creating a very basic program

Any group of Python statements placed sequentially in a file can be used as a program, or *script*. But it's more standard and useful to introduce additional structure. In its most basic form, this is a simple matter of creating a controlling function in a file and calling that function, as shown in listing 11.1.

Listing 11.1 File script1.py

```
def main():
    print("this is our first test script file")    <-- Controlling function main
main()                                              <-- Calls main
```

In this script, `main` is our controlling—and only—function. It's first defined, and then it's called. Although it doesn't make much difference in a small program, using this structure can give you more options and control when you create larger applications, so it's a good idea to make it a habit from the beginning.

11.1.1 Starting a script from a command line

If you're using Linux/UNIX, make sure Python is on your path and you're in the same directory as your script. Then type the following on your command line to start the script:

```
python script1.py
```

If you're using a Macintosh computer running OS X, the procedure is the same as for other UNIX systems. You need to open a terminal program, which is in the Utilities folder of the Applications folder. There are several other options for running scripts on OS X, which we'll discuss shortly.

If you're using Windows, open Command Prompt (in the Accessories subfolder of the All Programs folder in the Start menu). This opens in your home folder, and if necessary you can use the `cd` command to change to a subdirectory. For example, running script1.py if it was saved on your desktop would look like this:

```
C:\Documents and Settings\vern> cd Desktop                        <-- Changes to Desktop folder

C:\Documents and Settings\vern\Desktop> python script1.py         <-- Runs scriptl.py
hello                                                             <-- Output of scriptl.py

C:\Documents and Settings\vern\Desktop>
```

We'll be looking at other options for calling scripts later in this chapter, but stick with this for now.

11.1.2 Command-line arguments

A simple mechanism is available for passing in command-line arguments, as shown in listing 11.2.

Listing 11.2 File script2.py

```
import sys
def main():
    print("this is our second test script file")
    print(sys.argv)
main()
```

If we call this with the line

```
python script2.py arg1 arg2 3
```

we get

```
this is our second test script file
['script2.py', 'arg1', 'arg2', '3']
```

You can see that the command-line arguments have been stored in sys.argv as a list of strings.

11.1.3 Redirecting the input and output of a script

You can redirect the input and/or the output for a script using command-line options. To show this, we'll use the short script in listing 11.3.

Listing 11.3 File replace.py

```
import sys
def main():
   contents = sys.stdin.read()                                      # Reads from stdin into contents
   sys.stdout.write(contents.replace(sys.argv[1], sys.argv[2]))     # Replaces first argument with second
main()
```

This script reads its standard input and writes to its standard output whatever it read, with all occurrences of its first argument replaced with its second argument. Called as follows, it will place in `outfile` a copy of `infile` with all occurrences of `zero` replaced by `0`:

```
python replace.py zero 0 < infile > outfile
```

Note that this will work on UNIX; but on Windows, redirection of input and/or output will work only if you start a script from a command prompt window.

In general, the line

```
python script.py arg1 arg2 arg3 arg4 < infile > outfile
```

will have the effect of having any `input` or sys.stdin operations directed out of infile and any print or sys.stdout operations directed into outfile. The effect is as if you set sys.stdin to infile with `'r'` (read) mode and sys.stdout to outfile with `'w'` (write):

```
python replace.py a A < infile >> outfile
```

This line causes the output to be appended to outfile rather than overwrite it, as happens in the previous example.

You can also *pipe* in the output of one command as the input of another command:

```
python replace.py 0 zero < infile | python replace.py 1 one > outfile
```

This results in outfile containing the contents of infile, with all occurrences of `0` changed to `zero` and all occurrences of `1` changed to `one`.

11.1.4 The optparse module

You can configure a script to accept command-line options as well as arguments. The `optparse` module provides support for parsing different types of options and arguments and can even generate usage messages.

To use the `optparse` module, you create an instance of `OptionParser`, populate it with options, and then read both the options and the arguments. The example in listing 11.4 illustrates its use.

Listing 11.4 File opts.py

```
from optparse import OptionParser

def main():
    parser = OptionParser()
    parser.add_option("-f", "--file", dest="filename",                  ❶
                  help="write report to FILE", metavar="FILE")
    parser.add_option("-x", "--xray", dest="xray",
                  help="specify xray strength factor")
    parser.add_option("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,
                  help="don't print status messages to stdout")

    (options, args) = parser.parse_args()

    print("options:", str(options))
    print("arguments:", args)
main()
```

This adds a filename option with either `'-f'` or `'—file'` ❶. The final option added, the `"quiet"` option, also adds the ability to turn off the verbose option, which is `True` by default (`action=store_false`).

The `parse_args` method returns two objects: one containing the options and their values as key/value pairs, and a list of the positional arguments. You can get the values of the options from the first object by using the string you specified with the `dest` parameter as a dictionary key. If there's no argument for an option, the value is `None`. Any elements left on the command line after all the options have been parsed will be in the list of positional arguments. Thus, if you call the previous script with the line

```
python opts.py -x100 -q -f infile arg1 arg2      <-- Options come after script name
```

the following output will result:

```
options: {'xray': '100', 'verbose': False, 'filename': 'infile'}
arguments: ['arg1', 'arg2']
```

If an invalid option is found, or if an option that requires an argument isn't given one, `parse_args` raises an error.

```
python opts.py -x100 -r
```

This line results in the following response:

```
Usage: opts.py [options]

opts.py: error: no such option: -r
```

11.1.5 Using the fileinput module

The `fileinput` module is also sometimes useful for scripts. It provides support for processing lines of input from one or more files. It automatically reads the command-line arguments (out of sys.argv) and takes them as its list of input files. It allows you to then sequentially iterate through these lines. The simple example script in listing 11.5 (which will strip out any lines starting with `##`) illustrates the module's basic use.

Listing 11.5 File script4.py

```
import fileinput
def main():
    for line in fileinput.input():
        if not line.startswith('##'):
            print(line, end="")
main()
```

Now, assume we have the data files shown in listings 11.6 and 11.7.

Listing 11.6 File sole1.tst

```
## sole1.tst: test data for the sole function
0 0 0
0 100 0
##
0 100 100
```

Listing 11.7 File sole2.tst

```
## sole2.tst: more test data for the sole function
12 15 0
##
100 100 0
```

Also assume that we make the following call:

```
python script4.py sole1.tst sole2.tst
```

We obtain the following result with the comment lines stripped out and the data from the two files combined:

```
0 0 0
0 100 0
0 100 100
12 15 0
100 100 0
```

If no command-line arguments are present, the standard input is all that is read. If one of the arguments is a hyphen (-), the standard input is read at that point.

The module provides a number of other functions. These allow you at any point to determine the total number of lines that have been read (`lineno`), the number of lines that have been read out of the current file (`filelineno`), the name of the current file (`filename`), whether this is the first line of a file (`isfirstline`), and/or whether standard input is currently being read (`isstdin`). You can at any point skip to the next file (`nextfile`) or close the whole stream (`close`). The short script in listing 11.8 (which combines the lines in its input files and adds file-start delimiters) illustrates how you can use these.

Listing 11.8 File script5.py

```
import fileinput
def main():
    for line in fileinput.input():
        if fileinput.isfirstline():
            print("<start of file {0}>".format(fileinput.filename()))
        print(line, end="")
main()
```

Using the call

```
python script5.py file1 file2
```

will result in the following (where the dotted lines indicate the lines in the original files):

```
<start of file file1>
.......................
.......................
<start of file file2>
.......................
.......................
```

Finally, if you call `fileinput.input` with an argument of a single filename or a list of filenames, they're used as its input files rather than the arguments in sys.argv. `fileinput.input` also has an inline option that leaves its output in the same file as its input while optionally leaving the original around as a backup file. See the documentation for a description of this last option.

11.2 Making a script directly executable on UNIX

If you're on UNIX, you can easily make a script directly executable. Add the following line to its top and change its mode appropriately (that is, `chmod +x replace.py`):

```
#! /usr/bin/env python
```

Note that if Python 3.x isn't your default version of Python, you may need to change the `python` above to `python3.1` or something similar to specify that you want to use Python 3.x instead of an earlier default version.

Then, if you place your script somewhere on your path (for example, in your bin directory), you can execute it regardless of the directory you're in by typing its name and the desired arguments:

```
replace.py zero 0 < infile > outfile
```

On UNIX, you'll have input and output redirection and, if you're using a modern shell, command history and completion.

If you're writing administrative scripts on UNIX, a number of library modules are available that you may find useful. These include `grp` for accessing the group database, `pwd` for accessing the password database, `resource` for accessing resource usage information, `syslog` for working with the syslog facility, and `stat` for working with information about a file or directory obtained from an `os.stat` call. You can find information on this in the *Python Library Reference*.

11.3 Scripts on Mac OS X

In many ways, Python scripts on Mac OS X behave the same way as they do on Linux/UNIX. You can run Python scripts from a terminal window exactly the same way as on any UNIX box. But on the Mac, you can also run Python programs from the Finder, either by dragging the script file to the Python Launcher app or by configuring Python Launcher as the default application for opening your script (or, optionally, all files with a .py extension.)

There are several options for using Python on a Mac. The specifics of all the options are beyond the scope of this book, but you can get a full explanation by going to the python.org website and checking out the Mac section of the "Using Python" section of the documentation for your version of Python. You should also see section 11.7, "Distributing Python applications," for more information on how to distribute Python applications and libraries for the Mac platform.

If you're interested in writing administrative scripts for Mac OS X, you should look at packages that bridge the gap between Apple's Open Scripting Architectures (OSA) and Python. Two such packages are `appscript` and `PyOSA`.

11.4 Script execution options in Windows

If you're on Windows, you have a number of options for starting a script that vary in their capability and ease of use. Unfortunately, none of them are as flexible or powerful as on Linux/UNIX.

11.4.1 Starting a script as a document or shortcut

The easiest way to start a script on Windows is to use its standard document-opening technique. When you installed Python, it should have registered the .py suffix to itself. Verify this by confirming that your .py files are shown with a stylized python icon. If you double-click any .py file, Python is automatically called with this file as its argument. It's also entered onto the Documents list on your Start menu. But you're not able to enter any arguments, and the command window in which the interpreter is opened will close as soon as the script exits. If you want to have the window stay up so you can read the output, you can place the following line at the bottom of your controlling function:

```
input("Press the Enter key to exit")
```

This will leave the window up until you press Enter. You can also query the user for any input data you might have desired on the command line. Your current working directory at startup will be the one where your Python interpreter is located (C:\Python31, for example).

If you don't want the interpreter window to open (for example, when you're starting a GUI program using Tkinter), you can give the file the suffix .pyw. This will cause it to be opened by pythonw.exe. But if you start a script this way, any output to stdout or stderr will be thrown away.

You have more flexibility and the ability to pass in more information to your script if you set it up as a Windows shortcut (see figure 11.1).

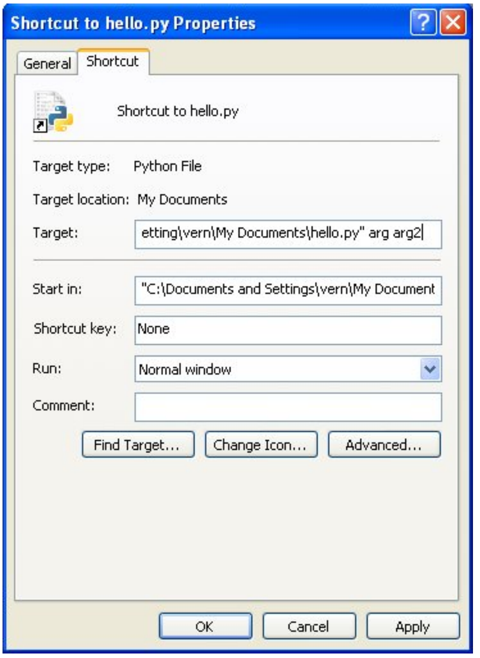


Figure 11.1 A Windows shortcut

Right-click the script, and select either the Create Shortcut or the Send to Desktop as Shortcut option. You can move the shortcut to any location and rename it as desired. By right-clicking it and selecting the Properties option, you can set the directory it starts in, type in arguments that it will be called with, and specify a shortcut key that will call it. The example in listing 11.9 illustrates this.

Listing 11.9 File script6.py

```
import sys, os
def main():
    print(os.getcwd())
    print(sys.argv)
    input("Hit return to exit")
main()
```

Creating the shortcut as shown in figure 11.1 and then calling this script (by pressing Ctrl-Alt-j or double-clicking its icon) brings up a Python window containing code similar to the following:

```
C:\Documents and Settings\vern\My Documents
['C:\\Documents and Settings\\vern\\My Documents\\script6.py']
Hit return to exit
```

The Python interpreter is implicitly called with the target line (because it has registered for Python files). You can also explicitly put it in. You may do this if you want to also enter options for the interpreter itself:

```
"C:\Python31\python.exe" -i
"C:\\Documents and Settings\\vern\\My Documents\\script6.py" arg1 arg2
```
[1]

Because you change the selection for the Run line to Minimized from the default Normal window, no MS-DOS window will be brought up, just as when you use the .pyw suffix for a document.

There is unfortunately no mechanism for redirecting the input or output for shortcuts.

11.4.2 Starting a script from the Windows Run box

It's possible to start scripts by placing the script on the desktop and typing

```
python script.py arg1 arg2
```

But as mentioned in the previous section, the output won't remain visible when the script ends. To see the output after the scripts ends, you similarly have to end it with an `input` line.

More flexibility is available with the Run box. Using the window that opens when you click the Browse button, you can search for scripts residing elsewhere (by selecting All Files for the Files of Type box at the bottom, because this defaults to Programs). This results in the pathname to the script you selected being displayed in the Run box (C:\My Documents\book\script1.py). You have to prefix this with *python*.

Also, clicking the selection button to the right of the text box brings up a history of your past lines of entries, from which you can select a line to edit. The current working directory for any script started from the Run box is the desktop.

11.4.3 Starting a script from a command window

To run a script from a command window, open a command prompt and navigate to the directory of your Python executable (C:\Python31 or wherever Python was installed). You can then enter your commands (the following examples assume that the scripts are in a scripts subfolder of the main Python folder):

```
python scripts\replace.py zero 0 < scripts\infile > scripts\outfile
```

You don't need to use the `-i` option because the script will run in your existing window.

---

[1] Please note that this should be entered as a single line with no line breaks on the target line.

This is the most flexible of the ways to run a script on Windows because it allows you to use input and output redirection. It's not all that convenient, because you don't have mouse editing, command completion, or the command history that you have in modern UNIX shells. It also doesn't provide the full power of an executable script on UNIX. It lacks the ability, for example, to navigate into a directory and call a script with a set of filenames from that directory as arguments without either having to place the script in that directory or enter the full pathname of the files or the script.

Note that if you're using input or output redirection, the default is for Windows to use the text mode. If you want your script to work with binary data, you either need to call the Python interpreter with the `-u` option set (`python -u script.py`) or set the environment variable `PYTHONUNBUFFERED=1` to turn off buffering and place it in a binary mode.

You can also add the directory where the Python interpreter is to your `PATH` environment variable. This frees you from having to move into the same directory as the interpreter or refer to it with a full pathname. The easiest way to do this on Windows XP is to right-click the My Computer desktop icon, choose Properties > Advanced, and then click the Environment Variables button. There, you can edit the `PATH` variable to add the path to the Python executable, usually something like c:\Python31.

11.4.4 Other Windows options

Other options are available to explore. If you're familiar with writing batch files, you can wrap your commands in them. A port of the GNU BASH shell comes with the Cygwin tool set, which you can read about at www.cygwin.com/. This provides a UNIX-like shell capability for Windows.

On Windows, you can edit the environment variables (see the previous section) to add .py as a magic extension, making your scripts automatically executable:

```
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.JS;.PY
```

11.5 Scripts on Windows vs. scripts on UNIX

The way you call scripts on Windows differs from the way scripts are called on Linux/UNIX, and that difference can affect what kind of scripts you develop and how you write them. Let's revisit our initial script from the section describing the `fileinput` module. Making it directly executable may be the only change we make to use it on UNIX (see listing 11.10).

Listing 11.10 File script4a.py

```
#! /usr/bin/env python3.1          <-- (1)
import fileinput
def main():
    for line in fileinput.input():
        if not line.startswith('##'):
            print(line, end="")
main()
```

This line allows the script to be executed on UNIX ❶. It has no effect on Windows. On UNIX, after this has been made executable and placed on your path, you're able to navigate into any desired directory and call this with wildcards and output redirection:

```
script4a.py sole*.tst > sole.data
```

This UNIX shell will find all files in the current directory that start with *sole* and end with *.tst*, and these will be sent in as command-line arguments.

On Windows, it's not so easy. As you've just discovered, using the convenient double-click mechanism, you can't pass in command-line arguments. Using the Run box, you can pass in arguments but not redirect the output. The closest you can come is to use the command prompt window: place the script in the desired directory, navigate into the directory, and then type[2]

```
C:\Python31\python.exe script4a.py sole1.tst sole2.tst
➥sole4.tst sole15.tst > sole.data
```

Or, use the following if you set your `PATH` as described in the previous section:

```
python script4a.py sole1.tst sole2.tst sole4.tst sole15.tst > sole.data
```

But using the `glob.glob` function, you can obtain the wildcard character functionality (see listing 11.11).

Listing 11.11 File script4b.py

```
#! /usr/bin/env python3.1
import fileinput, glob, sys, os
def main():
    if os.name == 'nt':                                  ❶
        sys.argv[1:] = glob.glob(sys.argv[1])            ❷
    for line in fileinput.input():
        if not line.startswith('##'):
            print(line, end="")
main()
```

This will be true if you're running under Windows XP, Vista, or Windows 7 ❶. Replace the single wildcarded filename string in the argument list with those files into which it expands ❷.

Assuming that we've placed our script in the same directory as our .tst files and navigated to that directory in a command prompt window, we can now call our script with a single wildcarded argument:

```
C:\Python31\python.exe script4b.py sole*.tst > sole.data
```

Or, again, if `PATH` has been modified:

```
python script4b.py sole*.tst > sole.data
```

[2] Please note that this should be entered as a single line with no line breaks. Due to space constraints, it could not be represented here in that manner.

With the use of copy and paste and storing the command text in a file, this might be acceptable for your situation. But the script can be set up so that it can be started using the double-click open mechanism. This is often done for scripts that will be run by end users.

When you start a script with a double-click, its working directory is the directory where the Python interpreter resides, not where the script is. Fortunately, the directory where the script is is appended as the first string in sys.path. The script in listing 11.12 uses this fact and changes into that directory. It then prompts the user for a possibly wildcarded input filename and expands it. Next, it prompts for the name of the output file and redirects standard output to this file. Finally, it proceeds to the original script's functionality.

Listing 11.12 File script4c.py

```
#! /usr/bin/env python3.1
import fileinput, glob, sys, os
def main():
    if os.name == 'nt':
        if sys.path[0]:
            os.chdir(sys.path[0])                          # Moves to appropriate directory
            input_filename = input("Name of input file:")
            input_list = glob.glob(input_filename)
            output_filename = input("Name of output file:")
            sys.stdout = open(output_filename, 'w')        # Redirects standard output
    else:
        input_list = sys.argv[1:]
    for line in fileinput.input(input_list):
       if not line.startswith('##'):
           print(line, end="")
main()
```

Placing this script in the directory of our .tst files, we can double-click it and enter the requested information. The result is the same as if it had been run from a command line:

```
Name of input file:sole*.tst
Name of output file:sole.data
```

This isn't as nice as on UNIX but in many cases may do fine.

11.6 Programs and modules

For small scripts that contain only a few lines of code, a single function works well. But if the script grows beyond this, separating your controlling function from the rest of the code is a good option to take. The rest of this section will illustrate this technique and some of its benefits. We'll start with an example using a simple controlling function. The script in listing 11.13 returns the English language name for a given number between 0 and 99.

Listing 11.13 File script7.py

```
#! /usr/bin/env python3.1
import sys
# conversion mappings
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
             '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
def num2words(num_string):
    if num_string == '0':
        return'zero'
    if len(num_string) > 2:
        return "Sorry can only handle 1 or 2 digit numbers"
    num_string = '0' + num_string                     <-- Pads on left in case it's a single-digit number
    tens, ones = num_string[-2], num_string[-1]
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def main():                                           <-- 1
    print(num2words(sys.argv[1]))
main()
```

If we call it with

```
python script7.py 59
```

we get this result:

```
fifty nine
```

Our controlling function here calls the function `num2words` with the appropriate argument and prints the result ❶. It's standard to have the call at the bottom, but sometimes you'll see the controlling function's definition at the top of the file. I prefer it at the bottom, just above the call, so I don't have to scroll back up to find it after going to the bottom to find out its name. This also cleanly separates the scripting plumbing from the rest of the file. This is useful when combining scripts and modules.

People combine scripts with modules when they want to make functions they've created in a script available to other modules or scripts. Also, a module may be instrumented so it can run as a script either to provide a quick interface to it for users or to provide hooks for automated module testing.

Combining a script and a module is a simple matter of putting the following conditional test around the controlling function:

```
if __name__ == '__main__':
    main()
else:
    # module-specific initialization code if any
```

If it's called as a script, it will be run with the name `__main__` and the controlling function, `main`, will be called. If it has been imported into an interactive session or another module, its name will be its filename.

When creating a script, I often set it as a module as well, right from the start. This allows me to import it into a session and interactively test and debug my functions as I create them. Only the controlling function needs to be debugged externally. If it grows, or I find myself writing functions I might be able to use elsewhere, I can separate those functions into their own module or have other modules import this module.

The script in listing 11.14 is an extension of the last script that has been set up to be able to be used as a module. The functionality has also been expanded to allow the entry of a number from 0 to 999999999999999 rather than just from 0 to 99. The controlling function (`main`) now does the checking of the validity of its argument and also strips out any commas in it, allowing more user-readable input like 1,234,567.

Listing 11.14 File n2w.py

```
#! /usr/bin/env python3.1
"""n2w: number to words conversion module: contains function
   num2words. Can also be run as a script
usage as a script: n2w num
           (Convert a number to its English word description)
           num: whole integer from 0 and 999,999,999,999,999 (commas are
           optional)
example: n2w 10,003,103
           for 10,003,103 say: ten million three thousand one hundred three
"""
import sys, string, optparse
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
             '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
_magnitude_list = [(0, ''), (3, ' thousand '), (6, ' million '),
                  (9, ' billion '), (12, ' trillion '),(15, '')]
def num2words(num_string):
    """num2words(num_string): convert number to English words"""
    if num_string == '0':
        return 'zero'
    num_length = len(num_string)
    max_digits = _magnitude_list[-1][0]
```

Usage message; includes example

Conversion mappings

Handles special conditions (number is zero or too large)

```
    if num_length > max_digits:
        return "Sorry, can't handle numbers with more than  " \
               "{0} digits".format(max_digits)
    num_string = '00' + num_string
    word_string = ''
    for mag, name in _magnitude_list:
        if mag >= num_length:
            return word_string
        else:
            hundreds, tens, ones = num_string[-mag-3], \
                 num_string[-mag-2], num_string[-mag-1]
            if not (hundreds == tens == ones == '0'):
                word_string = _handle1to999(hundreds, tens, ones) + \
                                                name + word_string
def _handle1to999(hundreds, tens, ones):
    if hundreds == '0':
        return _handle1to99(tens, ones)
    else:
        return _1to9dict[hundreds] + ' hundred ' + _handle1to99(tens, ones)
def _handle1to99(tens, ones):
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def test():
    values = sys.stdin.read().split()
    for val in values:
        num  = val.replace(',', '')
        print("{0} = {1}".format(val, num2words(num)))
def main():
    parser = optparse.OptionParser(usage=__doc__)
    parser.add_option("-t", "--test", dest="test",
                      action='store_true', default=False,
                      help="Test mode: reads from stdin")
    (options, args) = parser.parse_args()
    if options.test:
        test()
    else:
        if len(args) < 1:
            parser.error("incorrect number of arguments")
        num = sys.argv[1].replace(',', '')
        try:
            result = num2words(num)
        except KeyError:
            parser.error('argument contains non-digits')
        else:
            print("For {0}, say: {1}".format(sys.argv[1], result))
if __name__ == '__main__':
    main()
else:
    print("n2w  loaded as a module")
```

Pads number on left

Initiates string for number

Creates string containing number

Function for module test mode

Runs in test mode if test variable is set

Removes commas from number

Catches KeyErrors due to argument containing nondigits

1

If it's called as a script, the name will be `__main__`. If it's imported as a module, it will be named `n2w` **1**.

This `main` function illustrates the purpose of a controlling function for a command-line script, which, in effect, is to create a simplistic user interface for the user. It may handle the following tasks:

- Ensure that there is the right number of command-line arguments and that they're of the right types. Inform the user, giving usage information if not. Here, it ensures that there is a single argument, but it doesn't explicitly test to ensure that the argument contains only digits.
- Possibly handle a special mode. Here, a `'--test'` argument puts us in a test mode.
- Map the command-line arguments to those required by the functions, and call them in the appropriate manner. Here, commas are stripped out and the single function `num2words` is called.
- Possibly catch and print a more user-friendly message for exceptions that may be expected. Here, `KeyErrors` are caught, which will occur if the argument contains nondigits.[3]
- Map the output if necessary to a more user-friendly form. This is done here in the `print` statement. If this were a script to run on Windows, you would probably want to let the user open it with the double-click method—that is, to use the `input` to query for the parameter, rather than having it as a command-line option and keeping the screen up to display the output by ending the script with the line

  ```
  input("Press the Enter key to exit")
  ```

  But you may still want to leave the test mode in as a command-line option.

The test mode in listing 11.15 provides a regression test capability for the module and its `num2words` function. In this case, you use it by placing a set of numbers in a file.

Listing 11.15 File n2w.tst

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 98 99 100
101 102 900 901 999
999,999,999,999,999
1,000,000,000,000,000
```

Then, type

```
python n2w.py --test < n2w.tst > n2w.txt
```

The output file can be easily checked for correctness. This was run a number of times during its creation and can be rerun anytime `num2words` or any of the functions it calls are modified. And, yes, I'm aware that full exhaustive testing certainly didn't occur. I admit that well over 999 trillion valid inputs for this program haven't been checked!

---

[3] A better way to do this would be to explicitly check for nondigits in the argument using the regular expression module that will be introduced later. This would ensure that we don't hide `KeyErrors` that occur due to other reasons.

Often, the provision of a test mode for a module is the only function of a script. I know of at least one company where part of its development policy is to always create one for every Python module developed. Python's built-in data object types and methods usually make this easy, and those who practice this technique seem to be unanimously convinced that it's well worth the effort. See chapter 21 to find out more about testing your Python code.

Another option here would have been to create a separate file with just the portion of the `main` function that handles the argument and import `n2w` into this file. Then only the test mode would be left in the `main` function of `n2w.py`.

11.7 Distributing Python applications

You can distribute your scripts as source files (as .py files). You can also ship them as byte code (as .pyc or .pyo files). A byte code file will run under any platform as long as it has been written portably.

11.7.1 distutils

The standard way of packaging and distributing Python modules and applications is found in the `distutils` package. The details of creating a distribution are beyond the scope of this book, but if you want to package and distribute your Python code, the documentation on the python.org website has an extensive guide and a beginner's tutorial on the use of `distutils`. The heart of a `distutils` package is a setup.py file that you create, basically a Python program that controls the installation. `distutils` has a number of options and lets you create built distributions tailored for Windows and several UNIX platforms.

11.7.2 py2exe and py2app

Although it's not the purpose of this book to dwell on platform-specific tools, it's worth mentioning that `py2exe` creates Windows standalone programs and `py2app` does the same on the Mac OS X platform. By *standalone*, I mean they're single executables that can run on machines that don't have Python installed. In many ways, standalone executables aren't ideal, because they tend to be larger and less flexible than native Python applications; but in some situations they're the best, and sometimes the only, solution.

11.7.3 Creating executable programs with freeze

It's also possible to create an executable Python program that will run on machines that don't have Python installed by using the `freeze` tool. You'll find the instructions for this in the Readme file in the freeze directory in the Tools subdirectory of the Python source directory. If you're planning to use `freeze`, you'll probably need to download the Python source distribution.

In the process of "freezing" a Python program, you create C files, which are then compiled and linked using a C compiler, which you need to have installed on your sys-

tem. It will run only on the platform for which the C compiler you use provides its executables. Note that it will still be possible for someone to reverse-engineer your Python code from the resulting output. This also doesn't always work in a straightforward manner. This is one situation where you may end up needing to obtain support from the Python newsgroup, depending on the specifics of your application and which C compiler you're using. When you have your executable, it's generally robust.

11.8 Summary

Python scripts in their most basic form are sequences of Python statements placed in a file. A slightly more structured way to organize scripts that contain more than a few lines of Python code is presented here. This is a simple matter of using a control function to buffer out some of the control and interface logic. Modules can be instrumented to run as scripts, and scripts can be set up so they can be imported as modules. This provides a modular way to create large scripts and a simple mechanism to instrument a module with a regression test mode.

Scripts can be made executable on UNIX. They can be set up to support command-line redirection of their input and output, and with the `optparse` module it's easy to parse out complex combinations of command-line options and arguments.

On Mac OS X, you can run scripts from a terminal window in exactly the same way as on other UNIX platforms. In addition, you can use the Python Launcher to run Python programs, either individually or as the default application for opening Python files.

On Windows, you can call scripts a number of ways: by opening them with a double-click, using the Run window, or using a command-prompt window. It's possible to use command-line options and arguments in the last two of these and redirection in the last, but this isn't as convenient as on Linux/UNIX. Therefore, on Windows, command-line arguments and redirection are normally used only for special situations like test modes. This is also generally true for GUI programs on all three platforms.

Finally, as an alternative to scripts, `py2exe`, `py2app`, and the `freeze` tool provide the capability to provide an executable Python program that will run on machines that don't contain a Python interpreter.

Now that you have an understanding of the ways to create scripts and applications, the next step will be to look at how Python can interact with and manipulate filesystems.

12 Using the filesystem

This chapter covers

- Managing paths and pathnames
- Getting information about files
- Performing filesystem operations
- Processing all files in a directory subtree

Working with files involves one of two things: basic I/O (described in chapter 13, "Reading and writing files") and working with the filesystem (for example, naming, creating, moving, or referring to files), which is a bit tricky, because different operating systems have different filesystem conventions.

It would be easy enough to learn how to perform basic file I/O without learning all the features Python has provided to simplify cross-platform filesystem interaction—but I wouldn't recommend it. Instead, read at least the first part of this chapter. This will give you the tools you need to refer to files in a manner that doesn't depend on your particular operating system. Then, when you use the basic I/O operations, you can open the relevant files in this manner.

12.1 Paths and pathnames

All operating systems refer to files and directories with strings naming a given file or directory. Strings used in this manner are usually called *pathnames* (or sometimes just *paths*), which is the word we'll use for them. The fact that pathnames are strings introduces possible complications into working with them. Python does a good job of providing functions that help avoid these complications; but to make use of these Python functions effectively, you need an understanding of what the underlying problems are. This section discusses these details.

Pathname semantics across different operating systems are very similar, because the filesystem on almost all operating systems is modeled as a tree structure, with a disk being the root, and folders, subfolders, and so forth being branches, subbranches, and so on. This means that most operating systems refer to a specific file in fundamentally the same manner: with a pathname that specifies the path to follow from the root of the filesystem tree (the disk) to the file in question. (This characterization of the root corresponding to a hard disk is an oversimplification. But it's close enough to the truth to serve for this chapter.)

This pathname consists of a series of folders to descend into, in order to get to the desired file.

Different operating systems have different conventions regarding the precise syntax of pathnames. For example, the character used to separate sequential file or directory names in a Linux/UNIX pathname is /, whereas the character used to separate file or directory names in a Windows pathname is \. In addition, the UNIX filesystem has a single root (which is referred to by having a / character as the very first character in a pathname), whereas the Windows filesystem has a separate root for each drive, labeled A:\, B:\, C:\, and so forth (with C: usually being the main drive). Because of these differences, files will have different pathname representations on different operating systems. For example, a file called C:\data\myfile in MS Windows might be called /data/myfile on UNIX and on the Macintosh. Python provides functions and constants that allow you to perform common pathname manipulations without worrying about such syntactic details. With a little care, you can write your Python programs in such a manner that they will run correctly no matter what the underlying filesystem happens to be.

12.1.1 Absolute and relative paths

These operating systems allow two different types of pathnames. *Absolute* pathnames specify the exact location of a file in a filesystem, without any ambiguity; they do this by listing the entire path to that file, starting from the root of the filesystem. *Relative* pathnames specify the position of a file relative to some other point in the filesystem, and that other point isn't specified in the relative pathname itself; instead, the absolute starting point for relative pathnames is provided by the context in which they're used.

As examples of this, here are two Windows absolute pathnames:

C:\Program Files\Doom

A:\backup\June

And here are two Linux absolute pathnames and a Mac absolute pathname:

/bin/Doom

/floppy/backup/June

/Applications/Utilities

The following are two Windows relative pathnames:

mydata\project1\readme.txt

games\tetris

And these are two Linux/UNIX relative pathnames and one Mac relative pathname:

mydata/project1/readme.txt

games/tetris

Utilities/Java

Relative paths need context to anchor them. This is typically provided in one of two ways. The simplest is to append the relative path to an existing absolute path, producing a new absolute path. For example, we might have a relative Windows path, Start Menu\Programs\Explorer, and an absolute path, C:\Documents and Settings\Administrator. By appending the two, we have a new absolute path, C:\Documents and Settings\Administrator\Start Menu\Programs\Explorer, which refers to a specific file in the filesystem. By appending the same relative path to a different absolute path (say, C:\Documents and Settings\kmcdonald), we produce a path that refers to the Explorer program in a different user's (kmcdonald's) Profiles directory.

The second way in which relative paths may obtain a context is via an implicit reference to the *current working directory*, which is the particular directory where a Python program considers itself to be at any point during its execution. Python commands may implicitly make use of the current working directory when they're given a relative path as an argument. For example, if you use the `os.listdir(path)` command with a relative path argument, the anchor for that relative path is the current working directory, and the result of the command is a list of the filenames in the directory whose path is formed by appending the current working directory with the relative path argument.

12.1.2 The current working directory

Whenever you edit a document on a computer, you have a concept of where you are in that computer's file structure because you're in the same directory (folder) as the file you're working on. Similarly, whenever Python is running, it has a concept of where in the directory structure it is at any moment. This is important because the program may ask for a list of files stored in the current directory. The directory that a Python program is in is called the *current working directory* for that program. This may be different from the directory the program resides in.

To see this in action, start Python and use the `os.getcwd` (get current working directory) command to find out what Python's initial current working directory is:

```
>>> import os
>>> os.getcwd()
```

Note that os.getcwd is used as a zero-argument function call, to emphasize the fact that the value it returns isn't a constant but will change as you issue commands that change the value of the current working directory. (It will probably be either the directory the Python program itself resides in or the directory you were in when you started up Python. On my Linux machine, the result is /home/vceder, which is my home directory.) On Windows machines, you'll see extra backslashes inserted into the path—this is because Windows uses \ as its path separator, and in Python strings, as discussed earlier in section 6.3.1 on escape sequences, \ has a special meaning unless it's itself backslashed.

Now, type

```
>>> os.listdir(os.curdir)
```

The constant os.curdir returns whatever string your system happens to use as the same directory indicator. On both UNIX and Windows, this is a single dot; but to keep your programs portable, you should always use os.curdir instead of typing just the dot. This string is a relative path, meaning that os.listdir will append it to the path for the current working directory, giving the same path. This command returns a list of all of the files or folders inside the current working directory. Choose some folder folder, and type

```
>>> os.chdir(folder)        <-- "Change directory" function
>>> os.getcwd()
```

As you can see, Python moves into the folder specified as an argument of the os.chdir function. Another call to os.listdir(os.curdir) would return a list of files in folder, because os.curdir would then be taken relative to the new current working directory. Many Python filesystem operations (discussed later in this chapter) use the current working directory in this manner.

12.1.3 Manipulating pathnames

Now that you have the background to understand file and directory pathnames, it's time to look at the facilities Python provides for manipulating these pathnames. These facilities consist of a number of functions and constants in the os.path submodule, which you can use to manipulate paths without explicitly using any operating system–specific syntax. Paths are still represented as strings, but you need never think of them or manipulate them as such.

Let's start out by constructing a few pathnames on different operating systems, using the os.path.join function. Note that importing os is sufficient to bring in the os.path submodule also. There's no need for an explicit import os.path statement.

First, let's start Python under Windows:

```
>>> import os
>>> print(os.path.join('bin', 'utils', 'disktools'))
bin\utils\disktools
```

The `os.path.join` function interprets its arguments as a series of directory names or filenames, which are to be joined to form a single string understandable as a relative path by the underlying operating system. In a Windows system, that means path component names should be joined together with backslashes, which is what was produced.

Now, try the same thing in UNIX:

```
>>> import os
>>> print(os.path.join('bin', 'utils', 'disktools'))
bin/utils/disktools
```

The result is the same path, but using the Linux/UNIX convention of forward slash separators rather than the Windows convention of backslash separators. In other words, `os.path.join` lets you form file paths from a sequence of directory or filenames without any worry about the conventions of the underlying operating system. `os.path.join` is the fundamental way by which file paths may be built in a manner that doesn't constrain the operating systems on which your program will run.

The arguments to `os.path.join` need not be single a directory or filename; they may also be subpaths that are then joined together to make a longer pathname. The following example illustrates this in the Windows environment and is also a case where we find it necessary to use double backslashes in our strings. Note that we could enter the pathname with forward slashes (/) as well because Python converts them before accessing the Windows operating system:

```
>>> import os
>>> print(os.path.join('mydir\\bin', 'utils\\disktools\\chkdisk'))
mydir\bin\utils\disktools\chkdisk
```

Of course, if you always use `os.path.join` to build up your paths, you'll rarely need to worry about this. To write this example in a portable manner, we should enter

```
>>> path1 = os.path.join('mydir', 'bin');
>>> path2 = os.path.join('utils', 'disktools', 'chkdisk')
>>> print(os.path.join(path1, path2))
mydir\bin\utils\disktools\chkdisk
```

The `os.path.join` command also has some understanding of *absolute* versus *relative* pathnames. In Linux/UNIX, an absolute path always begins with a / (because a single slash denotes the topmost directory of the entire system, which contains everything else, including the various floppy and CD drives that might be available). A relative path in UNIX is any legal path that does *not* begin with a slash. Under any of the Windows operating systems, the situation is more complicated because the way in which MS Windows handles relative and absolute paths is messier. Rather than going into all of the details, I'll just say that the best way to handle this is to work with the following simplified rules for Windows paths:

- A pathname beginning with a drive letter followed by a backslash and then a path is an absolute path: C:\Program Files\Doom. (Note that C: by itself, without a trailing backslash, can't reliably be used to refer to the top-level directory on

the C: drive. You must use C:\ to refer to the top-level directory on C:. This is a result of DOS conventions, not Python design.)

- A pathname beginning with neither a drive letter nor a backslash is a relative path: mydirectory\letters\business.
- A pathname beginning with \\ followed by the name of a server is the path to a network resource.
- Anything else can be considered as an invalid pathname.[1]

Regardless of the operating system used, the `os.path.join` command doesn't perform sanity checks on the names it's constructing. It's possible to construct pathnames containing characters that, according to your OS, are forbidden in pathnames. If such checks are a requirement, probably the best solution is to write a small path-validity-checker function yourself.

The `os.path.split` command returns a two-element tuple splitting the basename of a path (the single file or directory name at the end of the path) from the rest of the path. For example, I use this on my Windows system:

```
>>> import os
>>> print(os.path.split(os.path.join('some', 'directory', 'path')))
('some\\directory', 'path')
```

The `os.path.basename` function returns only the basename of the path, and the `os.path.dirname` function returns the path up to but not including the last name. For example,

```
>>> import os
>>> os.path.basename(os.path.join('some', 'directory', 'path.jpg'))
'path.jpg'
>>> os.path.dirname(os.path.join('some', 'directory', 'path.jpg'))
'some\\directory'
```

To handle the dotted extension notation used by most filesystems to indicate file type (the Macintosh is a notable exception), Python provides `os.path.splitext`:

```
>>> os.path.splitext(os.path.join('some', 'directory', 'path.jpg'))
'some/directory/path', '.jpg')
```

The last element of the returned tuple contains the dotted extension of the indicated file (if there was a dotted extension.) The first element of the returned tuple contains everything from the original argument except the dotted extension.

You can also use more specialized functions to manipulate pathnames. `os.path.commonprefix(path1, path2, ...)` finds the common prefix (if any) for a set of paths. This is useful if you wish to find the lowest-level directory that contains every file in a set of files. `os.path.expanduser` expands username shortcuts in paths,

[1] Microsoft Windows allows some other constructs. But it's probably best to stick to the given definitions.

such as for UNIX. Similarly, `os.path.expandvars` does the same for environment variables., Here's an example on a Windows XP system:

```
>>> import os
>>> os.path.expandvars('$HOME\\temp')
'C:\\Documents and Settings\\administrator\\personal\\temp'
```

12.1.4 Useful constants and functions

You can access a number of useful path-related constants and functions to make your Python code more system independent than it otherwise would be. The most basic of these constants are `os.curdir` and `os.pardir`, which respectively define the symbol used by the operating system for the directory and parent directory path indicators. (In Windows as well as Linux/UNIX and Mac OS X, these are . and .. respectively.) These can be used as normal path elements; for example,

```
os.path.isdir(os.path.join(path, os.pardir, os.pardir))
```

asks if the parent of the parent of `path` is a directory. `os.curdir` is particularly useful for requesting commands on the current working directory. For example,

```
os.listdir(os.curdir)
```

returns a list of filenames in the current working directory (because `os.curdir` is a relative path, and `os.listdir` always takes relative paths as being relative to the current working directory).

The `os.name` constant returns the name of the Python module imported to handle the operating system–specific details. Here's an example on my Windows XP system:

```
>>> import os
>>> os.name
'nt'
```

Note that `os.name` returns `'nt'` even though the actual version of Windows is XP. Most versions of Windows, except for Windows CE, are identified as `'nt'`.

On a Mac running OS X and on Linux/UNIX, the response is `posix`. You can use this to perform special operations, depending on the platform you're working on:

```
import os
if os.name == 'posix':
    root_dir = "/"
elif os.name == 'nt':
    root_dir = "C:\\"
else:
    print("Don't understand this operating system!")
```

You may also see programs use `sys.platform`, which gives more exact information. On Windows XP, it's set to `win32`. On Linux, you may see `linux2`, whereas on Solaris, it may be set to `sunos5` depending on the version you're running.

All your environment variables, and the values associated with them, are available in a dictionary called `os.environ`; in most operating systems, this includes variables

related to paths, typically search paths for binaries and so forth. If what you're doing requires this, you know where to find it now.

At this point, you've received a grounding in the major aspects of working with pathnames in Python. If your immediate need is to open files for reading or writing, you can jump directly to the next chapter. Continue reading for further information about pathnames, testing what they point to, useful constants, and so forth.

12.2 Getting information about files

File paths are supposed to indicate actual files and directories on your hard drive. Of course, you're probably passing a path around because you wish to know something about what it points to. Various Python functions are available to do this.

The most commonly used Python path-information functions are `os.path.exists`, `os.path.isfile`, and `os.path.isdir`, which all take a single path as an argument. `os.path.exists` returns `True` if its argument is a path corresponding to something that exists in the filesystem. `os.path.isfile` returns `True` if and only if the path it's given indicates a normal data file of some sort (executables fall under this heading), and it returns `False` otherwise, including the possibility that the path argument doesn't indicate anything in the filesystem. `os.path.isdir` returns `True` if and only if its path argument indicates a directory; it returns `False` otherwise. These examples are valid on my system. You may need to use different paths on yours to investigate the behavior of these functions:

```
>>> import os
>>> os.path.exists('C:\\Documents and Settings\\vern\\My Documents')
True
>>> os.path.exists('C:\\Documents and Settings\\vern\\My
     Documents\\Letter.doc')
True
>>> os.path.exists('C:\\Documents and Settings\\vern\\\My
     Documents\\ljsljkflkjs')
False
>>> os.path.isdir('C:\\Documents and Settings\\vern\\My Documents')
True
>>> os.path.isfile('C:\\Documents and Settings\\vern\\My Documents')
False
>>> os.path.isdir('C:\\Documents and Settings\\vern\\My Documents
\\Letter.doc')
False
>>> os.path.isfile('C:\\Documents and Settings\\vern\\My
     Documents\\Letter.doc')
True
```

A number of similar functions provide more specialized queries. `os.path.islink` and `os.path.ismount` are useful in the context of Linux and other UNIX operating systems that provide file links and mount points. They return `True` if, respectively, a path indicates a file that's a link or a mount point. `os.path.islink` does *not* return `True` on Windows shortcuts files (files ending with .lnk), for the simple reason that such files aren't true links. The OS doesn't assign them a special status, and programs can't transpar-

ently use them as if they were the actual file. `os.path.samefile(path1, path2)` returns `True` if and only if the two path arguments point to the same file. `os.path.isabs(path)` returns `True` if its argument is an absolute path, `False` otherwise. `os.path.getsize(path)`, `os.path.getmtime(path)`, and `os.path.getatime(path)` return the size, last modify time, and last access time of a pathname, respectively.

12.3 More filesystem operations

In addition to obtaining information about files, Python lets you perform certain filesystem operations directly. This is accomplished through a set of basic but highly useful commands in the `os` module.

I'll describe only those true cross-platform operations. Many operating systems also have access to more advanced filesystem functions, and you'll need to check the main Python library documentation for the details.

You've already seen that to obtain a list of files in a directory, you use `os.listdir`:

```
>>> os.chdir(os.path.join('C:', 'my documents', 'tmp'))
>>> os.listdir(os.curdir)
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp']
```

Note that unlike the list directory command in many other languages or shells, Python does *not* include the `os.curdir` and `os.pardir` indicators in the list returned by `os.listdir`.

The `glob` function from the `glob` module (named after an old UNIX function that did pattern matching) expands Linux/UNIX shell-style wildcard characters and character sequences in a pathname, returning the files in the current working directory that match. A `*` matches any sequence of characters. A `?` matches any single character. A character sequence (`[h,H]` or `[0-9]`) matches any single character in that sequence:

```
>>> import glob
>>> glob.glob("*")
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp']
>>> glob.glob("*bkp")
['registry.bkp']
>>> glob.glob("?.tmp")
['a.tmp', '1.tmp', '7.tmp', '9.tmp']
>>> glob.glob("[0-9].tmp")
['1.tmp', '7.tmp', '9.tmp']
```

To rename (move) a file or directory, use `os.rename`:

```
>>> os.rename('registry.bkp', 'registry.bkp.old')
>>> os.listdir(os.curdir)
['book1.doc.tmp', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

You can use this command to move files across directories as well as within directories.

Remove or delete a data file with `os.remove`:

```
>>> os.remove('book1.doc.tmp')
>>> os.listdir(os.curdir)
'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

Note that you can't use `os.remove` to delete directories. This is a safety feature, to ensure that you don't accidentally delete an entire directory substructure by mistake.

Files can be created by writing to them, as you saw in the last chapter. To create a directory, use `os.makedirs` or `os.mkdir`. The difference between them is that `os.mkdir` doesn't create any necessary intermediate directories, but `os.makedirs` does:

```
>>> os.makedirs('mydir')
>>> os.listdir(os.curdir)
['mydir', 'a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
>>> os.path.isdir('mydir')
True
```

To remove a directory, use `os.rmdir`. This removes only empty directories. Attempting to use it on a nonempty directory raises an exception:

```
>>> os.rmdir('mydir')
>>> os.listdir(os.curdir)
['a.tmp', '1.tmp', '7.tmp', '9.tmp', 'registry.bkp.old']
```

To remove nonempty directories, use the `shutil.rmtree` function. It will recursively remove all files in a directory tree. See the appendix for the details of its use.

12.4 Processing all files in a directory subtree

Finally, a highly useful function for traversing recursive directory structures is the `os.walk` function. You can use it to walk through an entire directory tree, returning three things for each directory it traverses: the root, or path, of that directory; a list of its subdirectories; and a list of its files.

`os.walk` is called with the path of the starting, or top, folder and three optional arguments: `os.walk(directory, topdown=True, onerror=None, followlinks=False)`. `directory` is a starting directory path; if `topdown` is `True` or not present, the files in each directory are processed *before* its subdirectories, resulting in a listing that starts at the top and goes down; whereas if `topdown` is `True`, the subdirectories of each directory are processed **first**, giving a bottom-up traversal of the tree. The `onerror` parameter can be set to a function to handle any errors that result from calls to `os.listdir`, which are ignored by default. Finally, `os.walk` by default doesn't walk down into folders that are symbolic links, unless you give it the `followlinks=True` parameter.

When called, `os.walk` creates an iterator that recursively applies itself to all the directories contained in the `top` parameter. In other words, for each subdirectory `subdir` in `names`, `os.walk` recursively invokes a call to itself, of the form `os.walk(subdir, ...)`. Note that if `topdown` is `True` or not given, the list of subdirectories may be modified (using any of the list-modification operators or methods) before its items are used for the next level of recursion; you can use this to control into which—if any—subdirectories `os.walk` will descend.

To get a feel for `os.walk`, I recommend iterating over the tree and printing out the values returned for each directory. As an example of the power of `os.walk`, list the

current working directory and all of its subdirectories along with a count of the number of entries in each of them, excluding any .git directories:

```
import os
for root, dirs, files in os.walk(os.curdir):
    print("{0} has {1} files".format(root, len(files)))
    if ".git" in dirs:                                      <-- Checks for directory named .git
        dirs.remove(".git")                                 <-- Removes .git from directory list
```

This is complex, and if you want to use `os.walk` to its fullest extent, you should probably play around with it quite a bit to understand the details of what's going on.

The `copytree` function of the `shutil` module recursively makes copies of all the files in a directory and all of its subdirectories, preserving permission mode and stat (that is, access/modify times) information. It also has the already-mentioned `rmtree` function, for removing a directory and all of its subdirectories, as well as a number of functions for making copies of individual files. See the appendix for details.

12.5 Summary

Handling filesystem references (pathnames) and filesystem operations in a manner independent of the underlying operating system is never simple. Fortunately, Python provides a group of functions and constants that make this task much easier. For convenience, a summary of the functions discussed is given in table 12.1.

Table 12.1 Summary of filesystem values and functions

Function	Filesystem value or operation
`os.getcwd()`	Gets the current directory
`os.name`	Provides generic platform identification
`sys.platform`	Provides specific platform information
`os.environ`	Maps the environment
`os.listdir(path)`	Gets files in a directory
`os.chdir(path)`	Changes directory
`os.path.join(elements)`	Combines elements into a path
`os.path.split(path)`	Splits the path into a base and tail (the last element of the path)
`os.path.splitext(path)`	Splits the path into a base and a file extension
`os.path.basename(path)`	Gets the base of the path
`os.path.commonprefix(list_of_paths)`	Gets the common prefix for all paths on a list
`os.path.expanduser(path)`	Expands ~ or ~user to a full pathname
`os.path.expandvars(path)`	Expands environment variables

Table 12.1 Summary of filesystem values and functions *(continued)*

Function	Filesystem value or operation
`os.path.exists(path)`	Tests to see if a path exists
`os.path.isdir(path)`	Tests to see if a path is a directory
`os.path.isfile(path)`	Tests to see if a path is a file
`os.path.islink(path)`	Tests to see if a path is a symbolic link (not a Windows shortcut)
`os.path.ismount(path)`	Tests to see if a path is a mount point
`os.path.isabs(path)`	Tests to see if a path is an absolute path
`os.path.samefile(path_1, path_2)`	Tests to see if two paths refer to the same file
`os.path.getsize(path)`	Gets the size of a file
`os.path.getmtime(path)`	Gets the modification time
`os.path.getatime(path)`	Gets the access time
`os.rename(old_path, new_path)`	Renames a file
`os.mkdir(path)`	Creates a directory
`os.makedirs`	Creates a directory and any needed parent directories
`os.rmdir(path)`	Removes a directory
`glob.glob(pattern)`	Gets matches to a wildcard pattern
`os. walk(path)`	Gets all filenames in a directory tree

We didn't discuss more advanced filesystem operations that typically are tied to a certain operating system or systems here, so if your needs are more advanced and specialized, look at the main Python documentation for the `os` and `posix` modules.

Although being able to handle filesystem paths and operations is important, to write many programs you also need to be able to open, read, and write files. Those file operations are the subject of the next chapter.

13 Reading and writing files

This chapter covers

- Opening files and `file` objects
- Closing files
- Opening files in different modes
- Reading and writing text or binary data
- Redirecting screen input/output
- Using the struct module
- Pickling objects into files
- Shelving objects

Probably the single most common thing you'll want to do with files is open and read them.

13.1 Opening files and file objects

In Python, you open and read a file using the built-in `open` function and various built-in reading operations. The following short Python program reads in one line from a text file named myfile:

```
file_object = open('myfile', 'r')
line = file_object.readline()
```

open doesn't read anything from the file; instead it returns an object called a file object that you can use to access the opened file. A file object keeps track of a file and how much of the file has been read or written. All Python file I/O is done using file objects rather than filenames.

The first call to readline returns the first line in the file object, everything up to and including the first newline character or the entire file if there is no newline character in the file; the next call to readline would return the second line, and so on.

The first argument to the open function is a pathname. In the previous example, we're opening what we expect to be an existing file in the current working directory. The following opens a file at the given absolute location:

```
import os
file_name = os.path.join("c:", "My Documents", "test", "myfile")
file_object = open(file_name, 'r')
```

13.2 Closing files

After all data has been read from or written to a file object, it should be closed. Closing a file object frees up system resources, allows the underlying file to be read or written to by other code, and, in general, makes the program more reliable. For small scripts, not closing a file object generally doesn't have much of an effect; file objects are automatically closed when the script or program terminates. For larger programs, too many open file objects may exhaust system resources, causing the program to abort.

You close file objects using the close method, after the file object is no longer needed. The earlier short program then becomes this:

```
file_object = open("myfile", 'r')
line = file_object.readline()
# . . . any further reading on the file_object . . .
file_object.close()
```

13.3 Opening files in write or other modes

The second argument of the open command is a string denoting how the file should be opened. 'r' means open the file for reading, 'w' means open the file for writing (any data already in the file will be erased), and 'a' means open the file for appending (new data will be appended to the end of any data already in the file). If you want to open the file for reading, you can leave out the second argument; 'r' is the default. The following short program writes "Hello, World" to a file:

```
file_object = open("myfile", 'w')
file_object.write("Hello, World\n")
file_object.close()
```

Depending on the operating system, open may also have access to additional file modes. These aren't necessary for most purposes. As you write more advanced Python programs, you may wish to consult the Python reference manuals for details.

As well, open can take an optional third argument, which defines how reads or writes for that file are buffered. *Buffering* is the process of holding data in memory until enough has been requested or written to justify the time cost of doing a disk access. Other parameters to open control the encoding for text files and the handling of newline characters in text files. Again, these features aren't something you typically need to worry about, but as you become more advanced in your use of Python, you may wish to read up on them.

13.4 Functions to read and write text or binary data

I've presented the most common text file–reading function, readline. It reads and returns a single line from a file object, including any newline character on the end of the line. If there is nothing more to be read from the file, readline returns an empty string. This makes it easy to, for example, count the number of lines in a file:

```
file_object = open("myfile", 'r')
count = 0
while file_object.readline() != "":
    count = count + 1
print(count)
file_object.close()
```

For this particular problem, an even shorter way of counting all the lines is to use the built-in readlines method, which reads *all* the lines in a file and returns them as a list of strings, one string per line (with trailing newlines still included):

```
file_object = open("myfile", 'r')
print(len(file_object.readlines()))
file_object.close()
```

Of course, if you happen to be counting all the lines in a huge file, this may cause your computer to run out of memory, because it reads the entire file into memory at once. It's also possible to overflow memory with readline if you have the misfortune to try to read a line from a huge file that contains no newline characters, although this is highly unlikely. To handle such circumstances, both readline and readlines can take an optional argument affecting the amount of data they read at any one time. See the Python reference documentation for details.

Another way to iterate over all of the lines in a file is to treat the file object as an iterator in a for loop:

```
file_object = open("myfile", 'r')
count = 0
for line in file_object:
    count = count + 1
print(count)
file_object.close()
```

This method has the advantage that the lines are read into memory as needed, so even with large files, running out of memory isn't a concern. The other advantage of this method is that it's simpler and easier to read.

On some occasions, you may wish to read all the data in a file into a single `bytes` object, especially if the data isn't a string, and you want to get it all into memory so you can treat it as a byte sequence. Or you may wish to read data from a file as `bytes` objects of a fixed size. For example, you may be reading data without explicit newlines, where each line is assumed to be a sequence of characters of a fixed size. To do this, use the `read` method. Without any argument, it reads all the rest of a file and returns that data as a `bytes` object. With a single-integer argument, it reads that number of bytes, or less, if there isn't enough data in the file to satisfy the request, and returns a `bytes` object of the given size:

```
input_file = open("myfile", 'rb')
header = input_file.read(4)
data = input_file.read()
input_file.close()
```

The first line opens a file for reading in binary mode, the second line reads the first four bytes as a header string, and the third line reads the rest of the file as a single piece of data.

A possible problem with the `read` method may arise due to the fact that on Windows and Macintosh machines, text-mode translations occur if you use the `open` command in text mode—that is, without adding a `b` to the mode. In text mode, on a Macintosh any `\r` is converted to `"\n"`, whereas on Windows `"\r\n"` pairs are converted to `"\n"`. You can specify the treatment of newline characters by using the newline parameter when you open the file and specifying `newline="\n"`, `"\r"`, or `"\r\n"`, which forces only that string to be used as newline. If the file has been opened in binary mode, the newline parameter isn't needed—all bytes are returned exactly as they're in the file:

```
input_file = open("myfile", newline="\n")
```

This forces only `"\n"` to be considered a newline.

The converses of the `readline` and `readlines` methods are the `write` and `writelines` methods. Note that there is no `writeline` function. `write` writes a single string, which can span multiple lines if newline characters are embedded within the string—for example, something like

```
myfile.write("Hello")
```

`write` doesn't write out a newline after it writes its argument; if you want a newline in the output, you must put it there yourself. If you open a file in text mode (using `w`), any `\n` characters are mapped back to the platform-specific line endings (that is, `'\r\n'` on Windows or `'\r'` on Macintosh platforms). Again, opening the file with a specified newline will avoid this.

`writelines` is something of a misnomer; it doesn't necessarily write lines—it takes a list of strings as an argument and writes them, one after the other, to the given `file` object, without writing newlines. If the strings in the list end with newlines, they're written as lines; otherwise, they're effectively concatenated together in the file. But

`writelines` is a precise inverse of `readlines` in that it can be used on the list returned by `readlines` to write a file identical to the file `readlines` read from. For example, assuming myfile.txt exists and is a text file, this bit of code will create an exact copy of myfile.txt called myfile2.txt:

```
input_file = open("myfile.txt", 'r')
lines = input_file.readlines()
input_file.close()
output = open("myfile2.txt", 'w')
output.writelines(lines)
output.close()
```

13.4.1 Using binary mode

Sometimes, you want to access the data in a file as a string of bytes, with no translation or text encoding. To do this, you need to open files in *binary* mode, which will return a bytes object, instead of a string when reading. To open the file in binary mode, use the `'b'` (binary) argument with the mode—`open("file", 'rb')` or `open("file", 'wb')`:

```
input_file = open("myfile", 'rb')
header = input_file.read(4)
data = input_file.read()
input_file.close()
```

This example opens a file for reading, reads the first four bytes as a header string, and then reads the rest of the file as a single piece of data.

Keep in mind that files open in binary mode deal only in bytes, not strings. To use the data as strings, you must decode any `bytes` objects to `string` objects. This is often an important point in dealing with network protocols, where streams of data appear as files but are in binary mode as a rule.

13.5 Screen input/output and redirection

You can use the built-in `input` method to prompt for and read an input string:

```
>>> x = input("enter file name to use: ")
enter file name to use: myfile
>>> x
'myfile'
```

The prompt line is optional, and the newline at the end of the input line is stripped off.

To read in numbers using `input`, you need to explicitly convert the string that `input` returns to the correct number type. The following example uses `int`:

```
>>> x = int(input("enter your number: "))
enter your number: 39
>>> x
39
```

`input` writes its prompt to the *standard output* and reads from the *standard input*. Lower-level access to these and *standard error* can be had using the `sys` module, which

has sys.stdin, sys.stdout, and sys.stderr attributes. These can be treated as specialized file objects.

For sys.stdin, you have read, readline, and readlines methods. For sys.stdout and sys.stderr, you can use the standard print function as well as the write and writelines methods, which operate as they do for other file objects:

```
>>> import sys
>>> print("Write to the standard output.")
Write to the standard output.
>>> sys.stdout.write("Write to the standard output.\n")
Write to the standard output.
30
>>> s = sys.stdin.readline()
An input line
>>> s
'An input line\n'
```

sys.stdout.write returns number of characters written

You can redirect standard input to read from a file. Similarly, standard output or standard error can be set to write to files. They can also be subsequently programmatically restored to their original values using sys.__stdin__, sys.__stdout__, and sys.__stderr__:

```
>>> import sys
>>> f= open("outfile.txt", 'w')
>>> sys.stdout = f
>>> sys.stdout.writelines(["A first line.\n", "A second line.\n"])
>>> print("A line from the print statement")
>>> 3 + 4
>>> sys.stdout = sys.__stdout__
>>> f.close()
>>> 3 + 4
7
```

The print function also can be redirected to any file without changing standard output:

```
>>> import sys
>>> f = open("outfile.txt", 'w')
>>> print("A first line.\n", "A second line.\n", file=f)
>>> 3 + 4
7
>>> f.close()
>>> 3 + 4
7
```

While the standard output is redirected, you receive prompts and tracebacks from errors but no other output. If you're using IDLE, these examples using sys.__stdout__ won't work as indicated. You have to use the interpreter's interactive mode directly.

You'd normally use this when you're running from a script or program. But if you're using the interactive mode on Windows, you may want to temporarily redirect standard output in order to capture what might otherwise scroll off the screen. The short module in listing 13.1 implements a set of functions that provide this capability.

Listing 13.1 File mio.py

```
"""mio: module, (contains functions capture_output, restore_output,
     print_file, and clear_file )"""
import sys
_file_object = None
def capture_output(file="capture_file.txt"):
    """capture_output(file='capture_file.txt'): redirect the standard
    output to 'file'."""
    global _file_object
    print("output will be sent to file: {0}".format(file))
    print("restore to normal by calling 'mio.restore_output()'")
    _file_object = open(file, 'w')
    sys.stdout = _file_object

def restore_output():
    """restore_output(): restore the standard output back to the
             default (also closes the capture file)"""
    global _file_object
    sys.stdout = sys.__stdout__
    _file_object.close()
    print("standard output has been restored back to normal")

def print_file(file="capture_file.txt"):
    """print_file(file="capture_file.txt"): print the given file to the
         standard output"""
    f = open(file, 'r')
    print(f.read())
    f.close()

def clear_file(file="capture_file.txt"):
    """clear_file(file="capture_file.txt"): clears the contents of the
         given file"""
    f = open(file, 'w')
    f.close()
```

Here, `capture_output()` redirects standard output to a file that defaults to `"capture_file.txt"`. The function `restore_output()` restores standard output to the default. Also, `print_file()` prints this file to the standard output, and `clear_file()` clears its current contents.

13.6 Reading structured binary data with the struct module

Generally speaking, when working with your own files, you probably don't want to read or write binary data in Python. For very simple storage needs, it's usually best to use textual input and output as described earlier. For more sophisticated applications, Python provides the ability to easily read or write arbitrary Python objects (*pickling*, described later in this chapter). This ability is much less error prone than directly writing and reading your own binary data and is highly recommended.

But there's at least one situation in which you'll likely need to know how to read or write binary data, and that's when you're dealing with files that are generated or used

by other programs. This section gives a short description of how to do this using the `struct` module. Refer to the Python reference documentation for more details.

As you've seen, Python supports explicit binary input or output by using bytes instead of strings if you open the file in binary mode. But because most binary files rely on a particular structure to help parse the values, writing your own code to read and split them into variables correctly is often more work than it's worth. Instead, you can use the standard `struct` module to permit you to treat those strings as formatted byte sequences with some specific meaning.

Assume that we wish to read in a binary file called data, containing a series of records generated by a C program. Each record consists of a C short integer, a C double float, and a sequence of four characters that should be taken as a four-character string. We wish to read this data into a Python list of tuples, with each tuple containing an integer, a floating-point number, and a string.

The first thing to do is to define a *format string* understandable to the `struct` module, which tells the module how the data in one of our records is packed. The format string uses characters meaningful to `struct` to indicate what type of data is expected where in a record. For example, the character `'h'` indicates the presence of a single C short integer, and the character `'d'` indicates the presence of a single C double-precision floating-point number. Not surprisingly, `'s'` indicates the presence of a string and may be preceded by an integer to indicate the length of the string; `'4s'` indicates a string consisting of four characters. For our records, the appropriate format string is therefore `'hd4s'`. `struct` understands a wide range of numeric, character, and string formats. See the *Python Library Reference* for details.

Before we start reading records from our file, we need to know how many bytes to read at a time. Fortunately, `struct` includes a `calcsize` function, which takes our format string as an argument and returns the number of bytes used to contain data in such a format.

To read each record, we'll use the `read` method described previously. Then, the `struct.unpack` function conveniently returns a tuple of values by parsing a read record according to our format string. The program to read our binary data file is remarkably simple:

```
import struct
record_format = 'hd4s'
record_size = struct.calcsize(record_format)
result_list = []
input = open("data", 'rb')
while 1:
    record = input.read(record_size)        # Reads in single record
    if record == '':                        # (1)
        input.close()
        break
    result_list.append(struct.unpack(record_format, record))   # Unpacks record into tuple; appends to results
```

If the record is empty, we're at end of file, so we quit the loop (1). Note that there is no checking for file consistency. But if the last record is an odd size, the `struct.unpack` function raises an error.

As you may already have guessed, `struct` also provides the ability to take Python values and convert them into packed byte sequences. This is accomplished through the `struct.pack` function, which is almost, but not quite, an inverse of `struct.unpack`. The *almost* comes from the fact that whereas `struct.unpack` returns a tuple of Python values, `struct.pack` doesn't take a tuple of Python values; rather, it takes a format string as its first argument and then enough additional arguments to satisfy the format string. To produce a binary record of the form used in the previous example, we might do something like this:

```
>>> import struct
>>> record_format = 'hd4s'
>>> struct.pack(record_format, 7, 3.14, 'gbye')
b'\x07\x00\x00\x00\x1f\x85\xebQ\xb8\x1e\t@gbye'
```

`struct` gets even better than this; you can insert other special characters into the format string to indicate that data should be read/written in big-endian, little-endian, or machine-native-endian format (default is machine-native) and to indicate that things like C short integer should be sized either as native to the machine (the default) or as standard C sizes. If you need these features, it's nice to know they exist. See the *Python Library Reference* for details.

13.7 Pickling objects into files

Pickling is a *major* benefit in Python. Use this ability!

Python can write any data structure into a file and read that data structure back out of a file and re-create it, with just a few commands. This is an unusual ability but one that's highly useful. It can save you many pages of code that do nothing but dump the state of a program into a file (and can save a similar amount of code that does nothing but read that state back in).

Python provides this ability via the `pickle` module. Pickling is powerful but simple to use. For example, assume that the entire state of a program is held in three variables: `a`, `b`, and `c`. We can save this state to a file called state as follows:

```
import pickle
.
.
.
file = open("state", 'w')
pickle.dump(a, file)
pickle.dump(b, file)
pickle.dump(c, file)
file.close()
```

It doesn't matter what was stored in `a`, `b`, and `c`. It might be as simple as numbers or as complex as a list of dictionaries containing instances of user-defined classes. `pickle.dump` will save everything.

Now, to read that data back in on a later run of the program, just write

```
import pickle
file = open("state", 'r')
```

```
a = pickle.load(file)
b = pickle.load(file)
c = pickle.load(file)
file.close()
```

Any data that was previously in the variables `a`, `b`, or `c` is restored to them by `pickle.load`.

The `pickle` module can store almost anything in this manner. It can handle lists, tuples, numbers, strings, dictionaries, and just about anything made up of these types of objects, which includes all class instances. It also handles shared objects, cyclic references, and other complex memory structures correctly, storing shared objects only once and restoring them as shared objects, not as identical copies. But code objects (what Python uses to store byte-compiled code) and system resources (like files or sockets) can't be pickled.

More often than not, you won't want to save your entire program state with `pickle`. For example, most applications can have multiple documents open at one time. If you saved the entire state of the program, you would effectively save all open documents in one file. An easy and effective way of saving and restoring only data of interest is to write a save function that stores all data you wish to save into a dictionary and then uses `pickle` to save the dictionary. Then, you can use a complementary restore function to read the dictionary back in (again using `pickle`) and to assign the values in the dictionary to the appropriate program variables. This also has the advantage that there's no possibility of reading values back in an incorrect order—that is, an order different from the order in which they were stored. Using this approach with the previous example, we get code looking something like this:

```
import pickle
.
.
.
def save_data():
    global a, b, c
    file = open("state", 'w')
    data = {'a': a, 'b': b, 'c': c}
    pickle.dump(data, file)
    file.close()

def restore_data():
    global a, b, c
    file = open("state", 'r')
    data = pickle.load(file)
    file.close()
    a = data['a']
    b = data['b']
    c = data['c']
    .
    .
```

This is a somewhat contrived example. You probably won't be saving the state of the top-level variables of your interactive mode very often.

A real-life application is an extension of the cache example given in chapter 7, "Dictionaries." Recall that there, we were calling a function that performed a time-intensive calculation based on its three arguments. During the course of a program run, many of our calls to it ended up using the same set of arguments. We were able to obtain a significant performance improvement by caching the results in a dictionary, keyed by the arguments that produced them. But it was also the case that many different sessions of this program were being run many times over the course of days, weeks, and months. Therefore, by pickling the cache, we can keep from having to start over with every session. Listing 13.2 is a pared-down version of the module for doing this.

Listing 13.2 File sole.py

```
"""sole module: contains function sole, save, show"""
import pickle
_sole_mem_cache_d = {}
_sole_disk_file_s = "solecache"
file = open(_sole_disk_file_s, 'r')
_sole_mem_cache_d = pickle.load(file)
file.close()

def sole(m, n, t):
    """sole(m, n, t): perform the sole calculation using the cache."""
    global _sole_mem_cache_d
    if _sole_mem_cache_d.has_key((m, n, t)):
        return _sole_mem_cache_d[(m, n, t)]
    else:
        # . . . do some time-consuming calculations . . .
        _sole_mem_cache_d[(m, n, t)] = result
        return result

def save():
    """save(): save the updated cache to disk."""
    global _sole_mem_cache_d, _sole_disk_file_s
    file = open(_sole_disk_file_s, 'w')
    pickle.dump(_sole_mem_cache_d, file)
    file.close()

def show():
    """show(): print the cache"""
    global _sole_mem_cache_d
    print(_sole_mem_cache_d)
```

Initialization code executes when module loads

Public functions

This code assumes the cache file already exists. If you want to play around with it, use the following to initialize the cache file:

```
>>> import pickle
>>> file = open("solecache",'w')
>>> pickle.dump({}, file)
>>> file.close()
```

You'll also, of course, need to replace the comment `# . . . do some time-consuming calculations` with an actual calculation. Note that for production code, this is a

situation where you probably would use an absolute pathname for your cache file. Also, concurrency isn't being handled here. If two people run overlapping sessions, you'll end up with only the additions of the last person to save. If this were an issue, you could limit this overlap window significantly by using the dictionary update method in the save function.

13.8 Shelving objects

This is a somewhat advanced topic but certainly not a difficult one. This section is likely of most interest to people whose work involves storing or accessing pieces of data in large files, because the Python shelve module does exactly that—it permits the reading or writing of pieces of data in large files, without reading or writing the entire file. For applications that perform many accesses of large files (such as database applications), the savings in time can be spectacular. Like the pickle module (which it uses), the shelve module is simple.

Let's explore it through an address book. This sort of thing is usually small enough that an entire address file can be read in when the application is started and written out when the application is done. If you're an extremely friendly sort of person, and your address book is too big for this, it would be better to use shelve and not worry about it.

We'll assume that each entry in our address book consists of a tuple of three elements, giving the first name, phone number, and address of a person. Each entry will be indexed by the last name of the person the entry refers to. This is so simple that our application will be an interactive session with the Python shell.

First, import the shelve module and open the address book. shelve.open creates the address book file if it doesn't exist:

```
>>> import shelve
>>> book = shelve.open("addresses")
```

Now, add a couple of entries. Notice that we're treating the object returned by shelve.open as a dictionary (although it's a dictionary that can use only strings as keys):

```
>>> book['flintstone'] = ('fred', '555-1234', '1233 Bedrock Place')
>>> book['rubble'] = ('barney', '555-4321', '1235 Bedrock Place')
```

Finally, close the file and end the session:

```
>>> book..close()
```

So what? Well, in that same directory, start Python again, and open the same address book:

```
>>> import shelve
>>> book = shelve.open("addresses")
```

But now, instead of entering something, let's see if what we put in before is still around:

```
>>> book['flintstone']
('fred', '555-1234', '1233 Bedrock Place')
```

The addresses file created by `shelve.open` in the first interactive session has acted just like a persistent dictionary. The data we entered before was stored to disk, even though we did no explicit disk writes. That's exactly what `shelve` does.

More generally, `shelve.open` returns a `shelf` object that permits basic dictionary operations, key assignment or lookup, `del`, `in`, and the `keys` method. But unlike a normal dictionary, `shelf` objects store their data on disk, not in memory. Unfortunately, `shelf` objects do have one significant restriction as compared to dictionaries: they can use only strings as keys, versus the wide range of key types allowable in dictionaries.

It's important to understand the advantage `shelf` objects give you over dictionaries when dealing with large data sets. `shelve.open` makes the file accessible; it doesn't read an entire `shelf` object file into memory. File accesses are done only when needed, typically when an element is looked up, and the file structure is maintained in such a manner that lookups are very fast. Even if your data file is really large, only a couple of disk accesses will be required to locate the desired object in the file. This can improve your program in a number of ways. It may start faster, because it doesn't need to read a potentially large file into memory. It may execute faster, because more memory is available to the rest of the program, and thus less code will need to be swapped out into virtual memory. You can operate on data sets that are otherwise too large to fit in memory.

There are a few restrictions when using the `shelve` module. As previously mentioned, `shelf` object keys can be only strings; but any Python object that can be pickled can be stored under a key in a `shelf` object. Also, `shelf` objects aren't suitable for multiuser databases because they provide no control for concurrent access. Finally, make sure you close a `shelf` object when you've finished—this is sometimes required in order for changes you've made (entries or deletions) to be written back to disk.

As written, the cache example of the previous section would be an excellent candidate to be handled using shelves. You would not, for example, have to rely on the user to explicitly save their work to the disk. The only possible issue is that you wouldn't have the low-level control when you write back to the file.

13.9 Summary

File input and output in Python is a remarkably simple but powerful feature of the language. You can use various built-in functions to open, read, write, and close files. For very simple uses, you'll probably want to stick with reading and writing text, but the `struct` module does give you the ability to read or write packed binary data. Even better, the `pickle` and `shelve` modules provide simple, safe, and powerful ways of saving and accessing arbitrarily complex Python data structures, which means you may never again need to worry about defining file formats for your programs.

14 Exceptions

This chapter covers

- Understanding exceptions
- Handling exceptions in Python
- Using the `with` keyword

This chapter discusses exceptions, which are a language feature specifically aimed at handling unusual circumstances during the execution of a program. The most common use for exceptions is to handle errors that arise during the execution of a program, but they can also be used effectively for many other purposes. Python provides a comprehensive set of exceptions, and new ones can be defined by users for their own purposes.

The concept of exceptions as an error-handling mechanism has been around for some time. C and Perl, the most commonly used systems and scripting languages, don't provide any exception capabilities, and even programmers who use languages such as C++, which do include exceptions, are often unfamiliar with them. This chapter doesn't assume familiarity with exceptions on your part but instead provides detailed explanations. If you're already familiar with exceptions, you can skip directly to "Exceptions in Python" (section 14.2).

14.1 Introduction to exceptions

The following sections provide an introduction to exceptions and how they're used. Feel free to skip them if you're already familiar with exceptions from other languages.

14.1.1 General philosophy of errors and exception handling

Any program may encounter errors during its execution. For the purposes of illustrating exceptions, we'll look at the case of a word processor that writes files to disk and that therefore may run out of disk space before all of its data is written. There are various ways of coming to grips with this problem.

SOLUTION 1: DON'T HANDLE THE PROBLEM

The simplest way of handling this disk-space problem is to assume that there will always be adequate disk space for whatever files we write, and we needn't worry about it. Unfortunately, this seems to be the most commonly used option. It's usually tolerable for small programs dealing with small amounts of data, but it's completely unsatisfactory for more mission-critical programs. For several months while I was writing this book, my officemate spent hours every day cleaning up files that had become corrupt when a program written by someone else ran out of disk space and crashed. Eventually, he went into the code and put in some checks, which took care of most of the problem; even now, he still has to do occasional disk cleanups. Because the original program wasn't written cleanly with exceptions, the checks he put in could do only a partial job.

SOLUTION 2: ALL FUNCTIONS RETURN SUCCESS/FAILURE STATUS

The next level of sophistication in error handling is to realize that errors will occur and to define a methodology using standard language mechanisms for detecting and handling them. There are various ways of doing this, but a typical one is to have each function or procedure return a status value that indicates if that function or procedure call executed successfully. Normal results can be passed back in a call-by-reference parameter.

Let's look at how this might work with our hypothetical word-processing program. We'll assume that the program invokes a single high-level function, `save_to_file`, to save the current document to file. This will call various subfunctions to save different parts of the entire document to the file: for example, `save_text_to_file` to save the actual document text, `save_prefs_to_file` to save user preferences for that document, `save_formats_to_file` to save user-defined formats for the document, and so forth. Any of these may in turn call their own subfunctions, which save smaller pieces to the file. At the bottom will be built-in system functions, which write primitive data to the file and report on the success or failure of the file-writing operations.

We could put error-handling code into every function that might get a disk-space error, but that makes little sense. The only thing the error handler will be able to do is

to put up a dialog box telling the user that there's no more disk space and asking that the user remove some files and save again. It wouldn't make sense to duplicate this code everywhere we do a disk write. Instead, we'll put one piece of error-handling code into the main disk-writing function, `save_to_file`.

Unfortunately, for `save_to_file` to be able to determine when to call this error-handling code, every function it calls that writes to disk must itself check for disk space errors and return a status value indicating success or failure of the disk write. In addition, the `save_to_file` function must explicitly check every call to a function that writes to disk, even though it doesn't care about which function fails. The code, using a C-like syntax, looks something like this:

```
const ERROR = 1;
const OK = 0;
int save_to_file(filename) {
    int status;
    status = save_prefs_to_file(filename);
    if (status == ERROR) {
        ...handle the error...
    }
    status = save_text_to_file(filename);
    if (status == ERROR) {
        ...handle the error...
    }
    status = save_formats_to_file(filename);
    if (status == ERROR) {
        ...handle the error...
    }
    .
    .
    .
}
int save_text_to_file(filename) {
    int status;
    status = ...lower-level call to write size of text...
    if (status == ERROR) {
        return(ERROR);
    }
    status = ...lower-level call to write actual text data...
    if (status == ERROR) {
        return(ERROR);
    }
    .
    .
    .
}
```

And so on for `save_prefs_to_file`, `save_formats_to_file`, and all other functions that either write to `filename` directly or (in any way) call functions that write to `filename`.

Under this methodology, code to detect and handle errors can become a significant portion of the entire program, because every function and procedure containing

calls that might result in an error need to contain code to check for an error. Often, programmers don't have the time or the energy put in this type of complete error checking, and programs end up being unreliable and crash prone.

SOLUTION 3: THE EXCEPTION MECHANISM

It's obvious that most of the error-checking code in the previous type of program is largely repetitive: it checks for errors on each attempted file write and passes an error status message back up to the calling procedure if an error is detected. The disk space error is handled in only one place, the top-level `save_to_file`. In other words, most of the error-handling code is plumbing code, which connects the place where an error is generated with the place where it's handled. What we really want to do is to get rid of this plumbing and write code that looks something like this:

```
def save_to_file(filename)
    try to execute the following block
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    except that, if the disk runs out of space while
        executing the above block, do this
        ...handle the error...

def save_text_to_file(filename)
    ...lower-level call to write size of text...
    ...lower-level call to write actual text data...
    .
    .
    .
```

The error-handling code is completely removed from the lower-level functions; an error (if it occurs) will be generated by the built-in file writing routines and will propagate directly to the `save_to_file` routine, where our error-handling code will (presumably) take care of it. Although youcan't write this code in C, languages that offer exceptions permit exactly this sort of behavior; and, of course, Python is one such language. Exceptions let you write clearer code and handle error conditions better.

14.1.2 A more formal definition of exceptions

The act of generating an exception is called *raising* or *throwing* an exception. In the previous example, all exceptions are raised by the disk-writing functions, but exceptions can also be raised by any other functions or can be explicitly raised by your own code. We'll discuss this in more detail shortly. In the previous example, the low-level disk-writing functions (not seen in the code) would throw an exception if the disk were to run out of space.

The act of responding to an exception is called *catching* an exception, and the code that handles an exception is called *exception-handling code*, or just an *exception handler*.

In the example, the `except that...` line catches the disk-write exception, and the code that would be in place of the `...handle the error...` line would be an exception handler for disk-write (out of space) exceptions. There may be other exception handlers for other types of exceptions or even other exception handlers for the same type of exception but at another place in your code.

14.1.3 User-defined exceptions

Depending on exactly what event causes an exception, a program may need to take different actions. For example, an exception raised when disk space is exhausted needs to be handled quite differently from an exception that is raised if we run out of memory, and both are completely different from an exception that arises when a divide-by-zero error occurs. One way of handling these different types of exceptions would be to globally record an error message indicating the cause of the exception and to have all exception handlers examine this error message and take appropriate action. In practice, a different method has proven to be much more flexible.

Rather than defining a single kind of exception, Python, like most modern languages that implement exceptions, defines different types of exceptions, corresponding to various problems that may occur. Depending on the underlying event, different types of exceptions may be raised. In addition, the code that catches exceptions may be told to catch only certain types. This feature was used in the earlier pseudocode when we said, `except that, if the disk runs out of space . . ., do this`; we were specifying that this particular exception-handling code is interested only in disk-space exceptions. Another type of exception wouldn't be caught by that exception-handling code. It would either be caught by an exception handler that was looking for numeric exceptions, or, if there were no such exception handler, it would cause the program to exit prematurely with an error.

14.2 Exceptions in Python

The remaining sections of this chapter talk specifically about the exception mechanisms built into Python. The entire Python exception mechanism is built around an object-oriented paradigm, which makes it both flexible and expandable. If you aren't familiar with OOP, you don't need to learn OO techniques in order to use exceptions.

Like everything else in Python, an exception is an object. It's generated automatically by Python functions with a `raise` statement. After it's generated, the `raise` statement, which raises an exception, causes execution of the Python program to proceed in a manner different than would normally occur. Instead of proceeding with the next statement after the `raise`, or whatever generated the exception, the current call chain is searched for a handler that can handle the generated exception. If such a handler is found, it's invoked and may access the exception object for more information. If no suitable exception handler is found, the program aborts with an error message.

14.2.1 Types of Python exceptions

It's possible to generate different types of exceptions to reflect the actual cause of the error or exceptional circumstance being reported. Python provides a number of different exception types:

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError
            OverflowError
           ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EnvironmentError
            IOError
            OSError
                WindowsError (Windows)
                 VMSError (VMS)
        EOFError
        ImportError
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
            UnboundLocalError
        ReferenceError
        RuntimeError
            NotImplementedError
        SyntaxError
            IndentationError
                TabError
        SystemError
        TypeError
        ValueError
            UnicodeError
                UnicodeDecodeError
                UnicodeEncodeError
                UnicodeTranslateError
        Warning
            DeprecationWarning
             PendingDeprecationWarning
             RuntimeWarning
            SyntaxWarning
             UserWarning
             FutureWarning
             ImportWarning

            UnicodeWarning
             BytesWarningException
```

The Python exception set is hierarchically structured, as reflected by the indentation in this list of exceptions. As you saw in a previous chapter, you can obtain an alphabetized list from the `__builtin__` module.

Each type of exception is a Python class, which inherits from its parent exception type. But if you're not into OOP yet, don't worry about that. For example, an `IndexError` is also a `LookupError` and by inheritance an `Exception` and also a `BaseException`.

This hierarchy is deliberate: most exceptions inherit from `Exception`, and it's strongly recommended that any user-defined exceptions also subclass `Exception`, not `BaseException`. The reason is that if you have code set up like this

```
try:
    # do stuff
except Exception:
    # handle exceptions
```

you could still interrupt the code in the `try` block with Ctrl-C without triggering the exception-handling code, because the `KeyboardInterrupt` exception is *not* a subclass of `Exception`.

You can find an explanation of the meaning of each type of exception in the documentation, but you'll rapidly become acquainted with the most common types as you program!

14.2.2 Raising exceptions

Exceptions are raised by many of the Python built-in functions. For example:

```
>>> alist = [1, 2, 3]
>>> element = alist[7]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Error-checking code built into Python detects that the second input line requests an element at a list index that doesn't exist and raises an `IndexError` exception. This exception propagates all the way back to the top level (the interactive Python interpreter), which handles it by printing out a message stating that the exception has occurred.

Exceptions may also be raised explicitly in your own code, through the use of the `raise` statement. The most basic form of this statement is

```
raise exception(args)
```

The `exception(args)` part of the code creates an exception. The arguments to the new exception are typically values that aid you in determining what happened, something we'll discuss shortly. After the exception has been created, `raise` takes it and throws it upward along the stack of Python functions that were invoked in getting to the line containing the `raise` statement. The new exception is thrown up to the nearest (on the stack) exception catcher looking for that type of exception. If no catcher is found on the way to the top level of the program, this will either cause the program to terminate with an error or, in an interactive session, cause an error message to be printed to the console.

Try the following:

```
>>> raise IndexError("Just kidding")
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: Just kidding
```

The use of `raise` here generates what at first glance looks similar to all the Python list-index error messages you've seen so far. Closer inspection reveals this not to be the case. The actual error reported isn't as serious as those other ones.

The use of a string argument when creating exceptions is common. Most of the built-in Python exceptions, if given a first argument, assume it's a message to be shown to you as an explanation of what happened. This isn't always the case, though, because each exception type is its own class, and the arguments expected when a new exception of that class is created are determined entirely by the class definition. Also, programmer-defined exceptions, created by you or by other programmers, are often used for reasons other than error handling and, as such, may not take a text message.

14.2.3 Catching and handling exceptions

The important thing about exceptions isn't that they cause a program to halt with an error message. Achieving that in a program is never much of a problem. What's special about exceptions is that they don't have to cause the program to halt. By defining appropriate exception handlers, you can ensure that commonly encountered exceptional circumstances don't cause the program to fail; perhaps they display an error message to the user or do something else, even fix the problem, but they don't crash the program.

The basic Python syntax for exception catching and handling is as follows, using the `try` and `except` and sometimes the `else` keywords:

```
try:
    body
except exception_type1 as var1:
    exception_code1
except exception_type2 as var2:
    exception_code2
    .
    .
    .
except:
    default_exception_code
else:
    else_body
finally:
    finally_body
```

A `try` statement is executed by first executing the code in the `body` part of the statement. If this is successful (that is, no exceptions are thrown to be caught by the `try` statement), then the `else_body` is executed and the `try` statement is finished. Nothing else occurs. If an exception is thrown to the `try`, then the `except` clauses are

searched sequentially for one whose associated exception type matches that which was thrown. If a matching except clause is found, the thrown exception is assigned to the variable named after the associated exception type, and the exception code body associated with the matching exception is executed. If the line except exception_type, var: matches some thrown expression exc, the variable var will be created, and exc will be assigned as the value of var, before the exception-handling code of the except statement is executed. You don't need to put in var; you can say something like except exception_type:, which will still catch exceptions of the given type but won't assign them to any variable.

If no matching except clause is found, then the thrown exception can't be handled by that try statement, and the exception is thrown further up the call chain in hope that some enclosing try will be able to handle it.

The last except clause of a try statement can optionally refer to no exception types at all, in which case it will handle all types of exceptions. This can be convenient for some debugging and extremely rapid prototyping but generally isn't a good idea: all errors are hidden by the except clause, which can lead to some confusing behavior on the part of your program.

The else clause of a try statement is optional and is rarely used. It's executed if and only if the body of the try statement executes without throwing any errors.

The finally clause of a try statement is also optional and executes after the try, except, and else sections have executed. If an exception is raised in the try block and isn't handled by any of the except blocks, that exception is reraised after the finally block executes. Because the finally block always executes, it gives you a chance to include code to clean up after any exception handling by closing files, resetting variables, and so on.

14.2.4 Defining new exceptions

You can easily define your own exception. The following two lines will do this for you:

```
class MyError(Exception):
    pass
```

This creates a class that inherits everything from the base Exception class. But you don't have to worry about that if you don't want to.

You can raise, catch, and handle it like any other exception. If you give it a single argument (and you don't catch and handle it), this will be printed at the end of the traceback:

```
>>> raise MyError("Some information about what went wrong")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: Some information about what went wrong
```

This argument will, of course, be available to a handler you write as well:

```
try:
    raise MyError("Some information about what went wrong")
```

```
except MyError as error:
    print("Situation:", error)
```

The result here is

```
Situation: Some information about what went wrong
```

If you raise your exception with multiple arguments, these will be delivered to your handler as a tuple, which you can access through the `args` variable of the error:

```
try:
    raise MyError("Some information", "my_filename", 3)
except MyError as error:
    print("Situation: problem {0} with file {1}: {2}".format(
        (error.args[2],
 error.args[1], error.args[0]))
```

This gives the result

```
Situation: problem 3 with file my_filename: Some information
```

Because an exception type is a regular class in Python and happens to inherit from the root `Exception` class, it's a simple matter to create you own subhierarchy of exception types for use by your own code. You don't have to worry about this on a first read of the book. You can always come back to it after you've read chapter 15, "Classes and object-oriented programming." Exactly how you create your own exceptions depends on your particular needs. If you're writing a small program that may generate only a few unique errors or exceptions, subclass the main `Exception` class as we've done here. If, on the other hand, you're writing a large, multifile code library with a special goal in mind—say, weather forecasting—you may decide to define a unique class called `WeatherLibraryException` and then define all the unique exceptions of the library as subclasses of `WeatherLibraryException`.

14.2.5 Debugging programs with the assert statement

The `assert` statement is a specialized form of the `raise` statement:

```
assert expression, argument
```

The `AssertionError` exception with the optional `argument` is raised if the `expression` evaluates to `False` and the system variable `__debug__` is `True`. The `__debug__` variable defaults to `True`. It's turned off by either starting up the Python interpreter with the `-O` or `-OO` option or by setting the system variable `PYTHONOPTIMIZE` to `True`.

The code generator creates no code for assertion statements if `__debug__` is `false`. You can use `assert` statements to instrument your code with debug statements during development and leave them in the code for possible future use with no runtime cost during regular use:

```
>>> x = (1, 2, 3)
>>> assert len(x) > 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

14.2.6 The exception inheritance hierarchy

Now, let's expand on an earlier notion that Python exceptions are hierarchically structured and on what it means in terms of how `except` clauses catch exceptions.

The following code

```
try:
    body
except LookupError as error:
    exception code
except IndexError as error:
    exception code
```

catches two different types of exceptions: `IndexError` and `LookupError`. It just so happens that `IndexError` is a subclass of `LookupError`. If `body` throws an `IndexError`, that error is first examined by the `except LookupError as error:` line, and because an `IndexError` is a `LookupError` by inheritance, the first `except` will succeed. The second `except` clause will never be used because it's subsumed by the first `except` clause.

On the other hand, flipping the order of the two `except` clauses could potentially be useful; the first clause would then handle `IndexError` exceptions, and the second clause would handle any `LookupError` exceptions that aren't `IndexError` errors.

14.2.7 Example: a disk-writing program in Python

Let's revisit our example of a word-processing program that needs to check for disk out-of-space conditions as it writes a document to disk:

```
def save_to_file(filename) :
    try:
        save_text_to_file(filename)
        save_formats_to_file(filename)
        save_prefs_to_file(filename)
        .
        .
        .
    except IOError:
        ...handle the error...
def save_text_to_file(filename):
    ...lower-level call to write size of text...
    ...lower-level call to write actual text data...
    .
    .
    .
```

Notice how unobtrusive the error-handling code is; it's wrapped around the main sequence of disk-writing calls in the `save_to_file` function. None of the subsidiary disk-writing functions need any error-handling code. It would be easy to develop the program first and to add error-handling code later. That's often what is done, although this isn't the optimal ordering of events.

As another note of interest, this code doesn't respond specifically to disk-full errors; rather, it responds to `IOError` exceptions, which Python built-in functions raise

automatically whenever they can't complete an I/O request, for whatever reason. That's probably satisfactory for your needs; but if you need to identify disk-full conditions, you can do a couple of different things. The `except` body can check to see how much room is available on disk. If the disk is out of space, clearly it's a disk-full problem and should be handled in this `except` body; otherwise, the code in the `except` body can throw the `IOError` further up the call chain to be handled by some other `except`. If that isn't sufficient, you can do something more extreme, like going into the C source for the Python disk-writing functions and raising your own `DiskFull` exceptions as necessary. I wouldn't recommend this latter option, but it's nice to know the possibility exists if you need to make use of it.

14.2.8 Example: exceptions in normal evaluation

Exceptions are most often used in error handling but can also be remarkably useful in certain situations involving what we would think of as normal evaluation. An example I encountered involved a spreadsheet-like program I was implementing. Like most spreadsheets, it permits arithmetic operations involving cells, and it permits cells to contain values other than numbers. For my application, I wanted blank cells used in a numerical calculation to be considered as containing the value `0`, and cells containing any other nonnumeric string to be considered as invalid, which I represented as the Python value `None`. Any calculation involving an invalid value should return an invalid value.

The first step was to write a function that would evaluate a string from a cell of the spreadsheet and return an appropriate value:

```
def cell_value(string):
    try:
        return float(string)
    except ValueError:
        if string == "":
            return 0
        else:
            return None
```

Python's exception-handling ability made this a simple function to write. I tried to convert the string from the cell into a number and return it, in a `try` block using the `float` built-in function. `float` raises the `ValueError` exception if it can't convert its string argument to a number, so I caught that and returned either `0` or `None` depending on whether the argument string was empty or non-empty.

The next step was to handle the fact that some of my arithmetic might have to deal with a value of `None`. In a language without exceptions, the normal way to do this would be to define a custom set of arithmetic functions, which check their arguments for `None`, and then to use those functions rather than the built-in arithmetic functions to perform all of the spreadsheet arithmetic. This is time consuming and error prone, and it leads to slow execution because you're effectively building an interpreter in your spreadsheet. I took a different approach. All of my spreadsheet formulas were actually Python

functions that took as arguments the x and y coordinates of the cell being evaluated and the spreadsheet itself and calculated the result for the given cell using standard Python arithmetic operators, using `cell_value` to extract the necessary values from the spreadsheet. I defined a function called `safe_apply`, which took one of these formulas, applied it to the appropriate arguments in a `try` block, and returned either the formula's result or `None`, depending on whether the formula evaluated successfully:

```
def safe_apply(function, x, y, spreadsheet):
    try:
        return function, (x, y, spreadsheet)
    except TypeError:
        return None
```

These two changes were enough to integrate the idea of an empty (`None`) value into the semantics of my spreadsheet. Trying to develop this ability without the use of exceptions is a highly educational exercise.

14.2.9 Where to use exceptions

Exceptions are a natural choice for handling almost any error condition. It's an unfortunate fact that error handling is often added after the rest of the program is largely complete, but exceptions are particularly good at intelligibly managing this sort of after-the-fact error-handling code (or, more optimistically, for the case where you're adding more of them after the fact).

Exceptions are also highly useful in circumstances where a large amount of processing may need to be discarded after it has become obvious that a computational branch in your program has become untenable. The spreadsheet example was one such case; others are branch-and-bound algorithms and parsing algorithms.

14.3 Using with

Some situations, such as reading files, follow a predictable pattern with a set beginning and end. In the case of reading from a file, quite often the file needs to be open only one time, while data is being read, and then it can be closed. In terms of exceptions, you can code this kind of file access like this:

```
try:
    infile = open(filename)
    data = infile.read()
finally:
    infile.close()
```

In Python 3, there's a more generic way of handling situations like this: *context managers*. Context managers wrap a block and manage requirements on *entry* and *departure* from the block and are marked by the `with` keyword. File objects are context managers, and you can read files using that capability:

```
with open(filename) as infile:
    data = infile.read()
```

These two lines of code are equivalent to the five previous lines. In both cases, we know that the file will be closed immediately after the last read, whether the operation was successful or not. In the second case, closure of the file is also assured, because it's part of the file object's context management, so we don't need to write the code. In other words, by using `with` combined with a context management (in this case a file object), we don't need to worry about the routine cleanup.

Context managers are intended for things like locking and unlocking resources, closing files, committing database transactions, and so on. This is still a relatively new feature in Python, and its use will continue to be refined.

14.4 Summary

Python's exception-handling mechanism and exception classes provide a rich system to handle runtime errors in your code. Python's philosophy is that errors should not pass silently unless explicitly silenced, and by using `try`, `except`, `else`, and `finally` blocks and by selecting and even creating the types of exceptions caught, you can have very fine-grained control over how exceptions are handled and even ignored, when that makes sense.

You've seen that in Python exception types are organized in a hierarchy. That's possible because exceptions, like all objects in Python, are based on classes, and you'll see how Python uses classes and objects in the next chapter.

15 Classes and object-oriented programming

This chapter covers

- Defining classes
- Using instance variables and `@property`
- Defining methods
- Defining class variables and methods
- Inheriting from other classes
- Making variables and methods private
- Inheriting from multiple classes

In this chapter, we discuss Python classes, which can be used in a manner analogous to C structures but which can also be used in a full object-oriented manner. For the benefit of readers who aren't object-oriented programmers, we'll discuss the use of classes as structures in the first two subsections.

The remainder of the chapter discusses OOP in Python. This is only a description of the constructs available in Python; it's not an exposition on object-oriented programming itself.

15.1 Defining classes

A *class* in Python is effectively a data type. All the data types built into Python are classes, and Python gives you powerful tools to manipulate every aspect of a class's behavior. You define a class with the `class` statement:

```
class MyClass:
    body
```

`body` is a list of Python statements, typically variable assignments and function definitions. No assignments or function definitions are required. The body can be just a single `pass` statement.

By convention, class identifiers are in CapCase—that is, the first letter of each component word is capitalized, to make them stand out. After you define the class, a new object of the class type (an instance of the class) can be created by calling the class name as a function:

```
instance = MyClass()
```

15.1.1 Using a class instance as a structure or record

Class instances can be used as structures or records. Unlike C structures, the fields of an instance don't need to be declared ahead of time but can be created on the fly. The following short example defines a class called `Circle`, creates a `Circle` instance, assigns to the `radius` field of the circle, and then uses that field to calculate the circumference of the circle:

```
>>> class Circle:
...     pass
...
>>> my_circle = Circle()
>>> my_circle.radius = 5
>>> print(2 * 3.14 * my_circle.radius)
31.4
```

Like Java and many other languages, the fields of an instance/structure are accessed and assigned to by using dot notation.

You can initialize fields of an instance automatically by including an `__init__` initialization method in the class body. This function is run every time an instance of the class is created, with that new instance as its first argument. The `__init__` method is similar to a constructor in Java, but it doesn't really *construct* anything—it *initializes* fields of the class. This example creates circles with a radius of `1` by default:

```
class Circle:
    def __init__(self):                    ❶
        self.radius = 1                    ❷
my_circle = Circle()                       ❸
print(2 * 3.14 * my_circle.radius)         ❹
my_circle.radius = 5                       ❺
print(2 * 3.14 * my_circle.radius)
```

By convention, `self` is always the name of the first argument of `__init__`. `self` is set to the newly created circle instance when `__init__` is run ❶. Next, the code uses the class definition. We first create a `Circle` instance object ❷. The next line makes use of the fact that the radius field is already initialized ❸. We can also overwrite the radius field ❹; as a result, the last line prints a different result than the previous `print` statement ❺.

You can do a great deal more by using true object-oriented programming, and if you're not familiar with OOP, I urge you to read up on it. Python's object-oriented programming constructs are the subject of the remainder of this chapter.

15.2 Instance variables

Instance variables are the most basic feature of OOP. Take a look at the `Circle` class again:

```
class Circle:
    def __init__(self):
        self.radius = 1
```

`radius` is an *instance variable* of `Circle` instances. That is, each instance of the `Circle` class has its own copy of `radius`, and the value stored in that copy may be different from the values stored in the `radius` variable in other instances. In Python, you can create instance variables as necessary by assigning to a field of a class instance:

```
instance.variable = value
```

If the variable doesn't already exist, it's created automatically. This is how `__init__` creates the `radius` variable.

All uses of instance variables—both assignment and access—require *explicit mention* of the containing instance—that is, `instance.variable`. A reference to `variable` by itself is a reference not to an instance variable but rather to a local variable in the executing method. This is different from C++ or Java, where instance variables are referred to in the same manner as local method function variables. I rather like Python's requirement for explicit mention of the containing instance, because it clearly distinguishes instance variables from local function variables.

15.3 Methods

A *method* is a function associated with a particular class. You've already seen the special `__init__` method, which is called on a new instance when that instance is first created. In the following example, we define another method, `area`, for the `Circle` class, which can be used to calculate and return the area for any `Circle` instance. Like most user-defined methods, `area` is called with a *method invocation syntax* that resembles instance variable access:

```
>>> class Circle:
...     def __init__(self):
...         self.radius = 1
...     def area(self):
```

```
...         return self.radius * self.radius * 3.14159
...
>>> c = Circle()
>>> c.radius = 3
>>> print(c.area())
28.27431
```

Method invocation syntax consists of an instance, followed by a period, followed by the method to be invoked on the instance. The previous syntax is sometimes called *bound* method invocation. `area` can also be invoked as an *unbound* method by accessing it through its containing class. This is less convenient and is almost never done. When a method is invoked in this manner, its first argument must be an instance of the class in which that method is defined:

```
>>> print(Circle.area(c))
28.27431
```

Like `__init__`, the `area` method is defined as a function within the body of the class definition. The first argument of any method is the instance it was invoked by or on, named `self` by convention.

Methods can be invoked with arguments, if the method definitions accept those arguments. This version of `Circle` adds an argument to the `__init__` method, so that we can create circles of a given radius without needing to set the radius after a circle is created:

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * 3.14159
```

Note the two uses of `radius` here. `self.radius` is the instance variable called `radius`. `radius` by itself is the local function variable called `radius`. The two aren't the same! In practice, we'd probably call the local function variable something like `r` or `rad`, to avoid any possibility of confusion.

Using this definition of `Circle`, we can create circles of any radius with one call on the `circle` class. The following creates a `Circle` of radius 5:

```
c = Circle(5)
```

All the standard Python function features—default argument values, extra arguments, keyword arguments, and so forth—can be used with methods. For example, we could have defined the first line of `__init__` to be

```
    def __init__(self, radius=1):
```

Then, calls to `circle` would work with or without an extra argument; `Circle()` would return a circle of radius 1, and `Circle(3)` would return a circle of radius 3.

There's nothing magical about method invocation in Python. It can be considered shorthand for normal function invocation. Given a method invocation `instance.method(arg1, arg2, . . .)`, Python transforms it into a normal function call by using the following rules:

1 Look for the method name in the instance namespace. If a method has been changed or added for this instance, it's invoked in preference over methods in the class or superclass. This is the same sort of lookup discussed later in section 15.4.1.
2 If the method isn't found in the instance namespace, look up the class type `class` of `instance`, and look for the method there. In the previous examples, `class` is `Circle`—the type of the instance `c`.
3 If the method still isn't found, look for the method in the superclasses.
4 When the method has been found, make a direct call to it as a normal Python, using `instance` as the first argument of the function, and shifting all the other arguments in the method invocation one space over to the right. So, `instance.method(arg1, arg2, . . .)` becomes `class.method(instance, arg1, arg2, . . .)`.

15.4 Class variables

A *class variable* is a variable associated with a class, not an instance of a class, and is accessed by all instances of the class, in order to keep track of some class-level information, such as how many instances of the class have been created at any point in time. Python provides class variables, although using them requires slightly more effort than in most other languages. Also, you need to watch out for an interaction between class and instance variables.

A class variable is created by an assignment in the class body, not in the `__init__` function; after it has been created, it can be seen by all instances of the class. We can use a class variable to make a value for `pi` accessible to all instances of the `Circle` class:

```
class Circle:
    pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return self.radius * self.radius * Circle.pi
```

With the above definition entered, we can type

```
>>> Circle.pi
3.1415899999999999
>>> Circle.pi = 4
>>> Circle.pi
4
>>> Circle.pi = 3.14159
>>> Circle.pi
3.1415899999999999
```

This is exactly how we would expect a class variable to act. It's associated with and contained in the class that defines it. Notice in this example that we're accessing `Circle.pi` before any circle instances have been created. Obviously, `Circle.pi` exists independently of any specific instances of the `Circle` class.

You can also access a class variable from a method of a class, through the class name. We do so in the definition of `Circle.area`, where the `area` function makes specific reference to `Circle.pi`. In operation, this has the desired effect; the correct value for `pi` is obtained from the class and used in the calculation:

```
>>> c = Circle(3)
>>> c.area()
28.27431
```

You may object to hardcoding the name of a class inside that class's methods. You can avoid doing so through use of the special `__class__` attribute, available to all Python class instances. This attribute returns the class of which the instance is a member, for example:

```
>>> Circle
<class '__main__.Circle'>
>>> c.__class__
<class '__main__.Circle'>
```

The class named `Circle` is represented internally by an abstract data structure, and that data structure is exactly what is obtained from the `__class__` attribute of `c`, an instance of the `Circle` class. This lets us obtain the value of `Circle.pi` from `c` without ever explicitly referring to the `Circle` class name:

```
>>> c.__class__.pi
3.1415899999999999
```

Of course, we could use this internally in the `area` method to get rid of the explicit reference to the `Circle` class; replace `Circle.pi` with `self.__class__.pi`.

15.4.1 An oddity with class variables

There's a bit of an oddity with class variables that can trip you up if you aren't aware of it. When Python is looking up an instance variable, if it can't find an instance variable of that name, it will then try to find and return the value in a class variable of the same name. Only if it can't find an appropriate class variable will it signal an error. This does make it efficient to implement default values for instance variables; just create a class variable with the same name and appropriate default value, and avoid the time and memory overhead of initializing that instance variable every time a class instance is created. But this also makes it easy to inadvertently refer to an instance variable rather than a class variable, without signaling an error. Let's look at how this operates in conjunction with the previous example.

First, we can refer to the variable `c.pi`, even though `c` doesn't have an associated instance variable named `pi`. Python will first try to look for such an instance variable, but when it can't find it, it will then look for a class variable `pi` in `Circle` and find it:

```
>>> c = Circle(3)
>>> c.pi
3.1415899999999999
```

This may or may not be what you want; it's convenient but can be prone to error, so be careful.

Now, what happens if we attempt to use `c.pi` as a true class variable, by changing it from one instance with the intent that all instances should see the change? Again, we'll use the earlier definition for `Circle`:

```
>>> c1 = Circle(1)
>>> c2 = Circle(2)
>>> c1.pi = 3.14
>>> c1.pi
3.140000000000001
>>> c2.pi
3.1415899999999999
>>> Circle.pi
3.1415899999999999
```

This doesn't work as it would for a true class variable—`c1` now has its own copy of `pi`, distinct from the `Circle.pi` accessed by `c2`. This is because the assignment to `c1.pi` *creates* an instance variable in `c1`; it doesn't affect the class variable `Circle.pi` in any way. Subsequent lookups of `c1.pi` return the value in that instance variable, whereas subsequent lookups of `c2.pi` look for an instance variable `pi` in `c2`, fail to find it, and resort to returning the value of the class variable `Circle.pi`. If you want to change the value of a class variable, access it through the class name, not through the instance variable `self`.

15.5 Static methods and class methods

Python classes can also have methods that correspond explicitly to static methods in a language such as Java. In addition, Python has *class* methods, which are a bit more advanced.

15.5.1 Static methods

Just as in Java, you can invoke static methods even though no instance of that class has been created, although you *can* call them using a class instance. To create a static method, use the `@staticmethod` decorator, as shown in listing 15.1.

Listing 15.1 File circle.py

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    all_circles = []
    pi = 3.14159
    def __init__(self, r=1):
        """Create a Circle with the given radius"""
        self.radius = r
        self.__class__.all_circles.append(self)
    def area(self):
        """determine the area of the Circle"""
        return self.__class__.pi * self.radius * self.radius

    @staticmethod
    def total_area():
```

Variable containing list of all circles that have been created (annotation for `all_circles = []`)

```
        total = 0
        for c in Circle.all_circles:
            total = total + c.area()
        return total
```

Now, interactively type the following:

```
>>> import circle
>>> c1 = circle.Circle(1)
>>> c2 = circle.Circle(2)
>>> circle.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle.Circle.total_area()
31.415899999999997
```

Also notice that documentation strings are used. In a real module, you'd probably put in more informative strings, indicating in the class docstring what methods are available and including usage information in the method docstrings:

```
>>> circle.__doc__
'circle module: contains the Circle class.'
>>> circle.Circle.__doc__
'Circle class'
>>> circle.Circle.area.__doc__
'determine the area of the Circle'
```

15.5.2 Class methods

Class methods are similar to static methods in that they can be invoked before an object of the class has been instantiated or by using an instance of the class. But class methods are implicitly passed the class they belong to as their first parameter, so you can code them more simply, as in listing 15.2.

Listing 15.2 File circle_cm.py

```
"""circle module: contains the Circle class."""
class Circle:
    """Circle class"""
    all_circles = []          # Variable containing list of all circles that have been created
    pi = 3.14159
    def __init__(self, r=1):
        """Create a Circle with the given radius"""
        self.radius = r
        self.__class__.all_circles.append(self)
    def area(self):
        """determine the area of the Circle"""
        return self.__class__.pi * self.radius * self.radius

    @classmethod          # 1
    def total_area(cls):          # 2
        total = 0
        for c in cls.all_circles:          # 3
            total = total + c.area()
        return total
```

```
>>> import circle_cm
>>> c1 = circle_cm.Circle(1)
>>> c2 = circle_cm.Circle(2)
>>> circle_cm.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle_cm.Circle.total_area()
31.415899999999997
```

The `@classmethod` decorator is used before the method `def` ❶. The class parameter is traditionally `cls` ❷. You can use `cls` instead of `self.__class__` ❸.

By using a class method instead of a static method, we don't have to hardcode the class name into `total_area`. That means any subclasses of `Circle` can still call `total_area` and refer to their own members, not those in `Circle`.

15.6 Inheritance

Inheritance in Python is easier and more flexible than inheritance in compiled languages such as Java and C++ because the dynamic nature of Python doesn't force as many restrictions on the language.

To see how inheritance is used in Python, we start with the `Circle` class given previously and generalize. We might want to define an additional class for squares:

```
class Square:
    def __init__(self, side=1):
        self.side = side          # Length of any side of square
```

Now, if we want to use these classes in a drawing program, they must define some sense of where on the drawing surface each instance is. We can do so by defining an `x` coordinate and a `y` coordinate in each instance:

```
class Square:
    def __init__(self, side=1, x=0, y=0):
        self.side = side
        self.x = x
        self.y = y
class Circle:
    def __init__(self, radius=1, x=0, y=0):
        self.radius = radius
        self.x = x
        self.y = y
```

This approach works but results in a good deal of repetitive code as we expand the number of shape classes, because each shape will presumably want to have this concept of position. No doubt you know where we're going here. This is a standard situation for using inheritance in an object-oriented language. Instead of defining the `x` and `y` variables in each shape class, abstract them out into a general `Shape` class, and have each class defining an actual shape inherit from that general class. In Python, that looks like this:

```
class Shape:
    def __init__(self, x, y):
```

```
        self.x = x
        self.y = y
class Square(Shape):
    def __init__(self, side=1, x=0, y=0):
        super().__init__(x, y)
        self.side = side
class Circle(Shape):
    def __init__(self, r=1, x=0, y=0):
        super().__init__(x, y)
        self.radius = r
```

Says Square inherits from Shape

Must call __init__ method of Shape

Says Circle inherits from Shape

Must call __init__ method of Shape

There are (generally) two requirements in using an inherited class in Python, both of which you can see in the bolded code in the `Circle` and `Square` classes. The first requirement is defining the inheritance hierarchy, which you do by giving the classes inherited from, in parentheses, immediately after the name of the class being defined with the `class` keyword. In the previous code, `Circle` and `Square` both inherit from `Shape`. The second and more subtle element is the necessity to explicitly call the `__init__` method of inherited classes. Python doesn't automatically do this for you, but you can use the `super` function to have Python figure out which inherited class to use. This is accomplished in the example code by the `super().__init__(x,y)` lines. This calls the `Shape` initialization function with the instance being initialized and the appropriate arguments. If this weren't done, then in the example, instances of `Circle` and `Square` wouldn't have their `x` and `y` instance variables set.

Instead of using `super`, we could call `Shape`'s `__init__` by explicitly naming the inherited class using `Shape.__init__(self, x, y)`, which would also call the `Shape` initialization function with the instance being initialized. This wouldn't be as flexible in the long run, because it hardcodes the inherited class's name, which could be a problem later if the design and the inheritance hierarchy change. On the other hand, the use of `super` can be tricky in more complex cases. Because the two methods don't exactly mix well, clearly document whichever approach you use in your code.

Inheritance comes into effect when you attempt to use a method that isn't defined in the base classes but is defined in the superclass. To see this, let's define another method in the `Shape` class called `move`, which will move a shape by a given displacement. It will modify the `x` and `y` coordinates of the shape by an amount determined by arguments to the method. The definition for `Shape` now becomes

```
class Shape:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, delta_x, delta_y):
        self.x = self.x + delta_x
        self.y = self.y + delta_y
```

If we enter this definition for `Shape` and the previous definitions for `Circle` and `Square`, we can then engage in the following interactive session:

```
>>> c = Circle(1)
>>> c.move(3, 4)
```

```
>>> c.x
3
>>> c.y
4
```

If you try this and are doing it in an interactive session, be sure to reenter the `Circle` class after the redefinition of the `Shape` class.

The `Circle` class didn't define a `move` method immediately within itself, but because it inherits from a class that implements `move`, all instances of `Circle` can make use of `move`.

15.7 Inheritance with class and instance variables

Inheritance allows an instance to inherit attributes of the class. Instance variables are associated with object instances, and only one instance variable of a given name exists for a given instance.

To see this, consider the following example. Using these class definitions,

```
class P:
    z = "Hello"
    def set_p(self):
        self.x = "Class P"
    def print_p(self):
         print(self.x)
class C(P):
    def set_c(self):
        self.x = "Class C"
    def print_c(self):
        print(self.x)
```

execute the following code:

```
>>> c = C()
>>> c.set_p()
>>> c.print_p()
Class P
>>> c.print_c()
Class P
>>> c.set_c()
>>> c.print_c()
Class C
>>> c.print_p()
Class C
```

The object `c` in this example is an instance of class `C`. `C` inherits from `P`, but `c` doesn't inherit from some invisible instance of class `P`. It inherits methods and class variables directly from `P`. Because there is only one instance (`c`), any reference to the instance variable `x` in a method invocation on `c` must refer to `c.x`. This is true regardless of which class defines the method being invoked on `c`. As you can see, when they're invoked on `c`, both `set_p` and `print_p`, defined in class `P`, refer to the same variable referred to by `set_c` and `print_c` when they're invoked on `c`.

In general, this is what is desired for instance variables because it makes sense that references to instance variables of the same name should refer to the same variable. Occasionally, somewhat different behavior is desired, which you can achieve using private variables. These are explained in the next subsection.

Class variables are inherited, but you should take care to avoid name clashes and be aware of a generalization of the same behavior you saw in the earlier subsection on class variables. In our example, a class variable `z` is defined for the superclass `P`. It can be accessed in three different ways: through the instance `c`, through the derived class `C`, or directly through the superclass `P`:

```
>>> c.z; C.z; P.z
'Hello'
'Hello'
'Hello'
```

But if we try setting it through the class `C`, a new class variable will be created for the class `C`. This has no effect on `P`'s class variable itself (as accessed through `P`). But future accesses through the class `C` or its instance `c` will see this new variable rather than the original:

```
>>> C.z = "Bonjour"
>>> c.z; C.z; P.z
'Bonjour'
'Bonjour'
'Hello'
```

Similarly, if we try setting `z` through the instance `c`, a new instance variable will be created, and we'll end up with three different variables:

```
>>> c.z = "Ciao"
>>> c.z; C.z; P.z
'Ciao'
'Bonjour'
'Hello'
```

15.8 Private variables and private methods

A private variable or private method is one that can't be seen outside of the methods of the class in which it's defined. Private variables and methods are useful for a number of reasons. They enhance security and reliability by selectively denying access to important or delicate parts of an object's implementation. They avoid name clashes that can arise from the use of inheritance. A class may define a private variable and inherit from a class that defines a private variable of the same name, but this doesn't cause a problem, because the fact that the variables are private ensures that separate copies of them are kept. Finally, private variables make it easier to read code, because they explicitly indicate what is used only internally in a class. Anything else is the class's interface.

Most languages that define private variables do so through the use of a private or other similar keyword. The convention in Python is simpler, and it also makes it easier

to immediately see what is private and what isn't. Any method or instance variable whose name begins—but doesn't end—with a *double underscore* (__) is private; anything else isn't private.

As an example, consider the following class definition:

```
class Mine:
    def __init__(self):
        self.x = 2
        self.__y = 3        <-- Defines __y as private by using leading double underscores
    def print_y(self):
        print(self.__y)
```

Using this definition, create an instance of the class:

```
>>> m = Mine()
```

x isn't a private variable, so it's directly accessible:

```
>>> print(m.x)
2
```

__y is a private variable. Trying to access it directly raises an error:

```
>>> print(m.__y)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'Mine' object has no attribute '__y'
```

The `print_y` method isn't private, and because it's in the `Mine` class, it can access `__y` and print it:

```
>>> m.print_y()
3
```

Finally, you should note that the mechanism used to provide privacy is to *mangle* the name of private variables and private methods when the code is compiled to bytecode. What specifically happens is that `_classname` is appended to the variable name:

```
>>> dir(m)
['_Mine__y', 'x', ...]
```

The purpose is to avoid any accidental accesses. If someone wanted to, they could access the value. But by performing the mangling in this easily readable form, debugging is made easy.

15.9 Using @property for more flexible instance variables

Python allows you as the programmer to access instance variables directly, without the extra machinery of getter and setter methods often used in Java and other OO languages. This lack of getters and setters makes writing Python classes cleaner and easier; but in some situations, using getter and setter methods can be handy. Suppose you want a value before you put it into an instance variable or where it would be handy to

figure out an attribute's value on the fly. In both cases, getter and setter methods would do the job but at the cost of losing Python's easy instance variable access.

The answer is to use a property. A property combines the ability to pass access to an instance variable through methods like getters and setters and the straightforward access to instance variables through dot notation.

To create a property, you use the property decorator with a method having the property's name:

```
class Temperature:
    def __init__(self):
        self._temp_fahr = 0
    @property
    def temp(self):
        return (self._temp_fahr - 32) * 5 / 9
```

Without a setter, such a property is read-only. To change the property, you need to add a setter:

```
    @temp.setter
    def temp(self, new_temp):
        self._temp_fahr = new_temp * 9 / 5 + 32
```

Now, you can use standard dot notation to both get and set the property `temp`. Notice that the name of the method remains the same, but the decorator changes to the property name (`temp` in this case) plus `.setter` to indicate that a setter for the `temp` property is being defined:

```
>>> t = Temperature()
>>> t._temp_fahr
0
>>> t.temp
-17.77777777777779                    ❶
>>> t.temp = 34
>>> t._temp_fahr
93.200000000000003                    ❷
>>> t.temp
34.0
```

The `0` in `_temp_fahr` is converted to centigrade before it's returned ❶. The `34` is converted back to Fahrenheit by the setter ❷.

One big advantage of Python's ability to add properties is that you can do initial development with plain-old instance variables and then seamlessly change to properties whenever and wherever you need to without changing any client code—the access is still the same, using dot notation.

15.10 Scoping rules and namespaces for class instances

Now you have all the pieces to put together a picture of the scoping rules and namespaces for a class instance.

When you're in a method of a class, you have direct access to the *local namespace* (parameters and variables declared in the method), the *global namespace* (functions and variables declared at the module level), and the *built-in namespace* (built-in functions and built-in exceptions). These three namespaces are searched in the following order: local, global, and built in (see figure 15.1).

You also have access through the `self` variable to our *instance's namespace* (instance variables, private instance variables, and superclass instance variables), its *class's namespace* (methods, class variables, private methods, and private class variables), and its *superclass's namespace* (superclass methods and superclass class variables). These three namespaces are searched in the order instance, class, and then superclass (see figure 15.2).

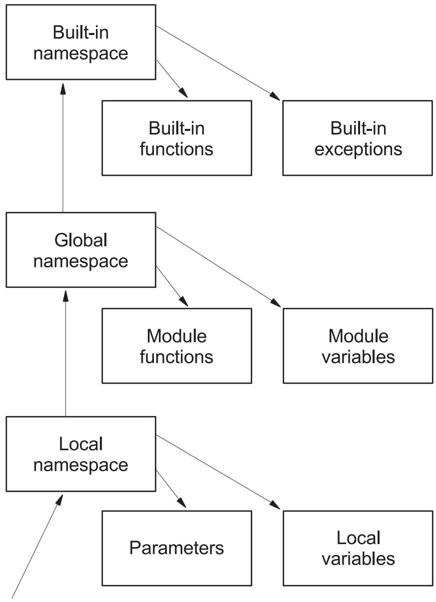


Figure 15.1 Direct namespaces

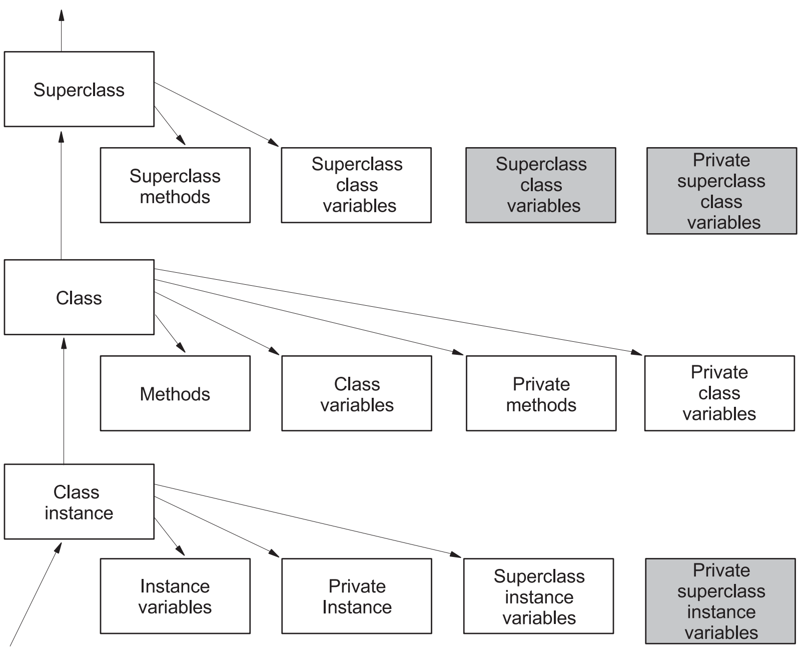


Figure 15.2 `Self` variable namespaces

Private superclass instance variables, private superclass methods, and private superclass class variables can't be accessed using `self`. A class is able to hide these names from its children.

The module in listing 15.3 puts these two together in one place to concretely demonstrate what can be accessed from within a method.

Listing 15.3 File cs.py

```
"""cs module: class scope demonstration module."""
mv ="module variable: mv"
def mf():
    return "module function (can be used like a class method in " \
           "other languages): mf()"
class SC:
    scv = "superclass class variable: self.scv"
    __pscv = "private superclass class variable: no access"
    def __init__(self):
        self.siv = "superclass instance variable: self.siv " \
                   "(but use SC.siv for assignment)"
        self.__psiv = "private superclass instance variable: " \
                      "no access"
    def sm(self):
        return "superclass method: self.sm()"
    def __spm(self):
        return "superclass private method: no access"
class C(SC):
    cv = "class variable: self.cv (but use C.cv for assignment)"
    __pcv = "class private variable: self.__pcv (but use C.__pcv " \
            "for assignment)"
    def __init__(self):
        SC.__init__(self)
        self.__piv = "private instance variable: self.__piv"
    def m2(self):
        return "method: self.m2()"
    def __pm(self):
        return "private method: self.__pm()"
    def m(self, p="parameter: p"):
        lv = "local variable: lv"
        self.iv = "instance variable: self.xi"
        print("Access local, global and built-in " \
              "namespaces directly")
        print("local namespace:", list(locals().keys()))
        print(p)                                            <- Parameter

        print(lv)                                 <- Instance variable
        print("global namespace:", list(globals().keys()))

        print(mv)                 <- Module variable

        print(mf())                               <- Module function
        print("Access instance, class, and superclass namespaces " \
```

```
        "through 'self'")
  print("Instance namespace:",dir(self))

  print(self.iv)                              Instance variable

  print(self.__piv)                           Private instance variable

  print(self.siv)                             Superclass instance variable
  print("Class namespace:",dir(C))
  print(self.cv)                              Class variable

  print(self.m2())                            Method

  print(self.__pcv)                           Private class variable

  print(self.__pm())                          Private module
  print("Superclass namespace:",dir(SC))
  print(self.sm())                            Superclass method

  print(self.scv)                             Superclass variable through instance
```

This output is considerable, so we'll look at it in pieces. In the first part, class `C`'s method `m`'s local namespace contains the parameters `self` (which is our instance variable) and `p` along with the local variable `lv` (all of which can be accessed directly):

```
>>> import cs
>>> c = cs.C()
>>> c.m()
Access local, global and built-in namespaces directly
local namespace: ['lv', 'p', 'self']
parameter: p
local variable: lv
```

Next, method `m`'s global namespace contains the module variable `mv` and the module function `mf`, (which, as described in a previous section, we can use to provide a class method functionality). There are also the classes defined in the module (the class `C` and the superclass `SC`). These can all be directly accessed:

```
global namespace: ['C', 'mf', '__builtins__', '__file__', '__package__',
     'mv', 'SC', '__name__', '__doc__']
module variable: mv
module function (can be used like a class method in other languages): mf()
```

Instance `C`'s namespace contains instance variable `iv` and our superclass's instance variable `siv` (which, as described in a previous section, is no different from our regular instance variable). It also has the mangled name of private instance variable `__piv` (which we can access through `self`) and the mangled name of our superclass's private instance variable `__psiv` (which we can't access):

```
Access instance, class, and superclass namespaces through 'self'
Instance namespace: ['_C__pcv', '_C__piv', '_C__pm', '_SC__pscv',
     '_SC__psiv', '_SC__spm', '__class__', '__delattr__', '__dict__',
     '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
     '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
```

```
    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
    '__weakref__', 'cv', 'iv', 'm', 'm2', 'scv', 'siv', 'sm']
instance variable: self.xi
private instance variable: self.__piv
superclass instance variable: self.siv (but use SC.siv for assignment)
```

Class `C`'s namespace contains the class variable `cv` and the mangled name of the private class variable `__pcv`: both can be accessed through `self`, but to assign to them we need to use class `C`. It also has the class's two methods `M` and `M2`, along with the mangled names of the private method `__PM` (which can be accessed through `self`):

```
Class namespace: ['_C__pcv', '_C__pm', '_SC__pscv', '_SC__spm', '__class__',
    '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
    '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
    '__subclasshook__', '__weakref__', 'cv', 'm', 'm2', 'scv', 'sm']
class variable: self.cv (but use C.cv for assignment)
method: self.m2()
class private variable: self.__pcv (but use C.__pcv for assignment)
private method: self.__pm()
```

Finally, superclass `SC`'s namespace contains superclass class variable `scv` (which can be accessed through `self`, but to assign to it we need to use the superclass `SC`) and superclass method `SM`. It also contains the mangled names of private superclass method `__SPM` and private superclass class variable `__spcv`, neither of which can be accessed through `self`:

```
Superclass namespace: ['_SC__pscv', '_SC__spm', '__class__', '__delattr__',
    '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
    '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
    '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
    '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
    '__subclasshook__', '__weakref__', 'scv', 'sm']
superclass method: self.SM()
superclass class variable: self.scv
```

This is a rather full example to decipher at first. You can use it as a reference or a base for your own exploration. As with most other concepts in Python, you can build a solid understanding of what is going on by playing around with a few simplified examples.

15.11 Destructors and memory management

You've already seen class constructors (the `__init__` methods). A destructor can be defined for a class as well. But unlike in C++, creating and calling a destructor isn't necessary to ensure that the memory used by your instance is freed. Python provides automatic memory management through a reference-counting mechanism. That is, it keeps track of the number of references to your instance; when this reaches zero, the memory used by your instance is reclaimed, and any Python objects referenced by your instance have their reference counts decremented by one. *For the vast majority of your classes, you won't need to define a destructor.*

C++ destructors are sometimes used to also perform cleanup tasks such as releasing or resetting system resources unrelated to memory management. To perform these functions in Python, using context managers using the `with` keyword or the definition of explicit cleanup or close methods for your classes is the best way to go.

If you need to, you *can* also define destructor methods for your classes. Python will implicitly call a class's destructor method `__del__` just before an instance is removed upon its reference count reaching zero. You can use this as a backup to ensure that your cleanup method is called. The following simple class illustrates this:

```
class SpecialFile:
    def __init__(self, file_name):
        self.__file = open(file_name, 'w')
        self.__file.write('***** Start Special File *****\n\n')
    def write(self, str):
        self.__file.write(str)
    def writelines(self, str_list):
        self.__file.writelines(str_list)
    def __del__(self):
        print("entered __del__")
        self.close()
    def close(self):
        if self.__file:
            self.__file.write('\n\n***** End Special File *****')
            self.__file.close()
            self.__file = None
```

Destructor method, __del__

Cleanup method, close

Notice that `close` is written so that it can be called more than once without complaint. This is what you'll generally want to do. Also, the `__del__` function has a print expression in it. But this is just for demonstration purposes. Take the following test function:

```
>>> def test():
...     f = SpecialFile('testfile')
...     f.write('111111\n')
...     f.close()
...
>>> test()
entered __del__
```

When the function `test` exits, `f`'s reference count goes to zero and `__del__` is called. Thus, in the normal case `close` is called twice, which is why we want `close` to be able to handle this. If we forgot the `f.close()` at the end of `test`, the file would still be closed properly because we're backed up by the call to the destructor. This also happens if we reassign to the same variable without first closing the file:

```
>>> f = SpecialFile('testfile')
>>> f = SpecialFile('testfile2')
entered __del__
```

As with the `__init__` constructor, the `__del__` destructor of a class's parent class needs to be called explicitly within a class's own destructor. Be careful when writing a destructor. If it's called when a program is shutting down, members of its global

namespace may already have been deleted. Any exception that occurs during its execution will be ignored, other than a message being sent of the occurrence to `sys.stderr`. Also, there's no guarantee that destructors will be called for all still-existing instances when the Python interpreter exits. Check the entries for destructors in the Python Language Manual and the Python FAQ for more details. They will also give you hints as to what may be happening in cases where you think all references to your object should be gone but its destructor hasn't been called.

Partly because of these issues, some people avoid using Python's destructors other than possibly to flag an error when they've missed putting in an explicit call. They prefer that cleanup always be done explicitly. Sometimes they're worth using, but only when you know the issues.

If you're familiar with Java, you're aware that this is what you have to do in that language. Java uses garbage collection, and its finalize methods aren't called if this mechanism isn't invoked (which may be never in some programs). Python's destructor invocation is more deterministic. When the references to an object go away, it's individually removed. On the other hand, if you have structures with cyclical references that have a `__del__` method defined, they aren't removed automatically. You have to go in and do this yourself. This is the main reason why defining your own `__del__` method destructors isn't recommended.

The following example illustrates the effect of a cyclical reference in Python and how you might break it. The purpose of the `__del__` method in this example is only to indicate when an object is removed:

```
>>> class Circle:
...     def __init__(self, name, parent):
...         self.name = name
...         self.parent = parent
...         self.child = None
...         if parent:
...             parent.child = self
...     def cleanup(self):
...         self.child = self.parent = None          # Breaks any cycles
...     def __del__(self):
...         print("__del__ called on", self.name)
...
>>> def test1():
...     a = Circle("a", None)
...     b = Circle("b", a)
...
>>> def test2():
...     c = Circle("c", None)
...     d = Circle("d", c)
...     d.cleanup()
...
>>> test1()                                          # 1
>>> test2()                                          # 2
__del__ called on c
__del__ called on d
```

Because they still refer to each other, `a` and `b` aren't removed when `test1` exits ❶. This is a memory leak. That is, each time `test1` is called, it leaks two more objects. The explicit call to the `cleanup` method is necessary to avoid this ❷.

The cycle is broken in the `cleanup` method, not the destructor, and we only had to break it in one place. Python's reference-counting mechanism took over from there. This approach is not only more reliable, but also more efficient, because it reduces the amount of work that the garbage collector has to do.

A more robust method of ensuring that our cleanup method is called is to use the `try-finally` compound statement. It takes the following form:

```
try:
    body
finally:
    cleanup_body
```

It ensures that `cleanup_body` is executed regardless of how or from where `body` is exited. We can easily see this by writing and executing another test function for the `Circle` class defined earlier:

```
>>> def test3(x):
...     try:
...         c = Circle("c", None)
...         d = Circle("d", c)
...         if x == 1:
...             print("leaving test3 via a return")
...             return
...         if x == 2:
...             print("leaving test3 via an exception")
...             raise RuntimeError
...         print("leaving test3 off the end")
...     finally:
...         d.cleanup()
...
>>> test3(0)
leaving test3 off the end
__del__ called on c
__del__ called on d
>>> test3(1)
leaving test3 via a return
__del__ called on c
__del__ called on d
>>> try:
...     test3(2)
... except RuntimeError:
...     pass
...
leaving test3 via an exception
__del__ called on c
__del__ called on d
```

Here, with the addition of three lines of code, we're able to ensure that our cleanup method is called when our function is left, which in this case can be via an exception, a return statement, or returning after its last statement.

15.12 Multiple inheritance

Compiled languages place severe restrictions on the use of multiple inheritance, the ability of objects to inherit data and behavior from more than one parent class. For example, the rules for using multiple inheritance in C++ are so complex that many people avoid using it. In Java, multiple inheritance is completely disallowed, although Java does have the interface mechanism.

Python places no such restrictions on multiple inheritance. A class can inherit from any number of parent classes, in the same way it can inherit from a single parent class. In the simplest case, none of the involved classes, including those inherited indirectly through a parent class, contains instance variables or methods of the same name. In such a case, the inheriting class behaves like a synthesis of its own definitions and all of its ancestor's definitions. For example, if class `A` inherits from classes `B`, `C`, and `D`, class `B` inherits from classes `E` and `F`, and class `D` inherits from class `G` (see figure 15.3), and none of these classes share method names, then an instance of class `A` can be used as if it were an instance of any of the classes `B`–`G`, as well as `A`; an instance of class `B` can be used as if it were an instance of class `E` or `F`, as well as class `B`; and an instance of class `D` can be used as if it were an instance of class `G`, as well as class `D`. In terms of code, the class definitions y look like this:

```
class E:
        . . .
class F:
        . . .
class G:
        . . .
class D(G):
        . . .
class C:
        . . .
class B(E, F):
        . . .
class A(B, C, D):
        . . .
```

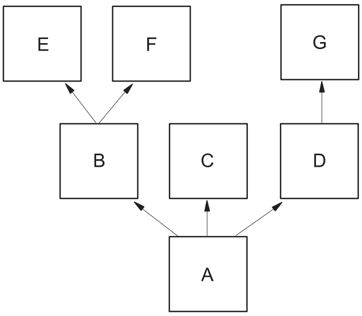


Figure 15.3 Inheritance hierarchy

The situation is more complex when some of the classes share method names, because Python must then decide which of the identical names is the correct one. For example, assume we wish to resolve a method invocation `a.f()`, on an instance `a` of class `A`, where `f` isn't defined in `A` but is defined in all of `F`, `C`, and `G`. Which of the various methods will be invoked?

The answer lies in the order in which Python searches base classes when looking for a method not defined in the original class on which the method was invoked. In the simplest cases, Python looks through the base classes of the original class in left-to-right order but always looks through all of the ancestor classes of a base class before looking in the next base class. In attempting to execute `a.f()`, the search goes something like this:

1 Python first looks in the class of the invoking object, class `A`.

2. Because `A` doesn't define a method `f`, Python starts looking in the base classes of `A`. The first base class of `A` is `B`, so Python starts looking in `B`.
3. Because `B` doesn't define a method `f`, Python continues its search of `B` by looking in the base classes of `B`. It starts by looking in the first base class of `B`, class `E`.
4. `E` doesn't define a method `f` and also has no base classes, so there is no more searching to be done in `E`. Python goes back to class `B` and looks in the next base class of `B`, class `F`.

Class `F` does contain a method `f`, and because it was the first method found with the given name, it's the method used. The methods called `f` in classes `C` and `G` are ignored.

Of course, using internal logic like this isn't likely to lead to the most readable or maintainable of programs. And with more complex hierarchies, other factors come into play to make sure that no class is searched twice and to support cooperative calls to `super`.

But this is probably a more complex hierarchy than you'd expect to see in practice. If you stick to the more standard uses of multiple inheritance, as in the creation of mixin or addin classes, you can easily keep things readable and avoid name clashes.

Some people have a strong conviction that multiple inheritance is a bad thing. It can certainly be misused, and nothing in Python forces you to use it. After being involved with a number of object-oriented project developments in industry since starting with one of the first versions of C++ in 1987, I've concluded that one of the biggest dangers seems to be creating inheritance hierarchies that are too deep. Multiple inheritance can at times be used to help keep this from happening. That issue is beyond the scope of this book. The example we use here only illustrates how multiple inheritance works in Python and doesn't attempt to explain the use cases—for example, as in mixin or addin classes—for it.

15.13 Summary

This chapter has briefly presented the basics of Python object-oriented programming in a way that will make it easy for any reader familiar with OO to instantly use these features of Python. We made no attempt to make this an introduction to OOP. If you need to learn the basic concepts, refer to OOP books on the market.

As you become more experienced with Python, you'll find you can also do deeper things than have been described here. In addition to the features described in this chapter, an aspiring Python OO programmer may also wish to use the operator-overloading features provided by special method attributes, which are described in chapter 20.

Classes and objects can be particularly useful in dealing with GUI interfaces, as you'll see in the next chapter, which explores creating cross-platform GUI applications with Python.

16 Graphical user interfaces

This chapter covers

- Installing Tkinter
- Starting and using Tkinter
- Understanding the principles of Tk
- Writing a simple Tkinter application
- Creating and placing widgets
- Alternatives to Tkinter

This chapter, an introduction to programming GUIs in Python, will do two things. First, it will provide a look at the GUI package that comes with Python, taking into account things like its ease of use, the capabilities of the package, cross-platform portability, and so forth. Second, it will give a brief overview of what else is available for GUI programming with Python and how to find it.

The Python core language has no built-in support for GUIs. It's a pure programming language, like C, Perl, or Pascal. As such, any support for GUIs must come from libraries external to Python, and many such libraries have been developed.

Of all the GUI packages currently available to Python programmers, Tkinter is the one commonly used. Tkinter is an object-oriented layer on top of the Tcl/Tk graphics libraries. The code that drives it is stable, efficient, and well supported.

Although it has been knocked for its somewhat plain appearance, in Python 3.1 Tkinter adds support for the new ttk widgets, which greatly improve its look and feel. I feel that it's a good choice for developing GUIs in Python for several reasons:

- *It's well integrated into Python.*
  Because IDLE uses Tkinter, it's included in many distributions of Python with no extra installation or can be added easily. That means that you (and your users) don't have as many dependencies to worry about if you want to distribute your application.
- *It's extremely powerful.*
  Complex GUIs can be coded in a short period of time and with a small amount of code.
- *It's a true cross-platform GUI.*
  When you learn it on any supported platform (currently, Windows, Macintosh, and almost all variants of Linux and UNIX), you can transfer all your knowledge directly to all other supported platforms. If you use the ttk widgets, even the look and feel of the native GUI is supported.
- *Tkinter, like Python, is free.*
  You can use it widely throughout your organization without worries about cost.

If you decide that Tkinter isn't for you, or you already have enough experience with Tkinter to know this, look at the section near the end of this chapter, which covers other possible GUI solutions.

Tk vs. Tkinter

In this section, we're talking about both the Tk GUI extension to the Tcl language and Tkinter, the Python library that uses Tk and wraps the Tk interface in Python classes. In general, we'll refer to Tkinter, because that's all that a Python programmer will usually touch directly. Sometimes we'll refer to Tk directly, particularly when talking about general features of the underlying Tk library.

Tkinter contains a huge number of features. The following sections aren't a lesson on how to use Tkinter but more of an introductory overview, primarily aimed at readers who aren't familiar with Tkinter and want to know if it's worth their while to look further into it. Remember, for every feature I mention, there are dozens more that I don't. The Tk command reference alone is almost 300 pages, and that doesn't cover any of the basic concepts.

16.1 Installing Tkinter

If you're using IDLE, you already have Tkinter installed. On Windows and Mac OS X, Tkinter comes as part of the Python distribution. Some Linux distributions don't come

with Tkinter by default, but they usually have Tkinter packages available. On Ubuntu Linux, for example, installing the IDLE package also installs Tkinter. If you don't have Tkinter installed, or are having problems making it run, go to the Python home page and search for "tkinter" to find the link to the Tkinter page, which contains documentation, tutorials, troubleshooting information, and instructions on how download the latest version for your platform. Don't be confused by the fact that you'll be installing something probably called Tcl/Tk, followed by some version number. Tcl is a scripting language, and Tk is a GUI extension. Python uses Tcl to access Tk, but in a transparent fashion; you'll never need to worry about the Tcl aspect of the package.

16.2 Starting Tk and using Tkinter

After you've installed Tkinter on your system, check to make sure everything is working properly. Start up Python and type

```
from tkinter import *
```

If you receive another Python command prompt >>> and no errors, then everything is working okay, and Tk has been started automatically by the importation of Tkinter.

If you'd like to see a brief example of Tkinter in action, the following code creates a dialog box like the one in figure 16.1:

```
import sys
win = Tk()
button = Button(win, text="Goodbye", command=sys.exit)
button.pack()
mainloop()
```



Figure 16.1 A minimal Tkinter application

Note that if you're in IDLE, you may need to omit the last line. See the sidebar on `mainloop()` and IDLE.

Note about mainloop() and IDLE

If you want to run the Tkinter examples in this chapter from within IDLE, you need to be aware that IDLE has two modes: one that allows modules to be run in subprocesses and one that doesn't. If your version of IDLE runs in the subprocess mode, which is the default on Windows and Mac OS X, you can run the code in this chapter as is, with the `mainloop()` line included. You should also leave that line in if you're using the command line or running from the emacs mode.

The no-subprocess mode can be recognized by the `==== No Subprocess ====` message that appears in the shell window when IDLE starts; this is the default in some Linux distributions, such as Ubuntu 9.10. If you run IDLE from a command line, the no-subprocess mode is triggered by adding the `-n` parameter. If IDLE is running in no-subprocess mode, omit the final line containing `mainloop()`, or put a # in front of it to make it a comment. In no-subprocess mode, IDLE is already running a `mainloop` under Tk, and running a second `mainloop` may cause the whole IDLE process to hang.

When you click the Goodbye button, the `sys.exit` command is executed, and Python quits. This window is only a little larger than the button it contains and may appear behind another window. But as long as you didn't get any error messages, it should be there.

16.3 Principles of Tkinter

The Tkinter GUI package is based on a small number of basic principles and ideas, and this is the main reason it's relatively easy to learn and use. Although it certainly helps to have some previous knowledge of GUI-based programming, this isn't strictly necessary. Tkinter is a relatively easy way for you to learn GUI and event-driven programming.

16.3.1 Widgets

The first basic idea behind Tkinter is the concept of a *widget*, which is short for *window gadget*. A widget is a data structure that also has a visible, onscreen representation. When the program changes the internal data structure of the widget, that change is automatically displayed on the screen. Various user actions on the visible representation of the widget (mouse clicks and so forth) can, in turn, cause internal changes or actions within the widget's data structure.

Tkinter is a collection of widget definitions, together with commands for operating on them, and a few extra commands that don't apply to any specific widget but that are still relevant to GUI programming. In Python, each different type of widget is represented by a different Python class. A `Button` widget is of the `Button` class, a `Label` widget is of the `Label` class, and so forth.

This direct mapping between Tkinter widget types and Python classes makes using widgets in a Python program extremely simple. For example, a Python program that creates and uses a `Button` widget and a `Label` widget looks something like this:

```
from tkinter import *

...
my_button = Button(...optional arguments...)
my_label = Label(...optional arguments...)
...
```

This style of mapping Python classes to widgets is common in Python GUI environments, although the exact names of the widgets and their parameters will naturally vary.

16.3.2 Named attributes

The second basic idea behind Tkinter is the availability and use of *named attributes* to fine-tune widget behavior. To understand why this is necessary and see how useful it is, let's look at an apparently simple task—creating a button.

The simplest way to do this is to specify a class, say `AButton`, with a one-argument object constructor, whose single argument is a string that will become the name displayed on the button. Creating a button looks like this:

```
my_button = AButton(name)
```

But this provides no way to associate a command with the button—that is, the name of a function that should be executed when the user clicks the button. To do this, change `AButton` to have a two-argument constructor, with the command as the second argument:

```
my_button = AButton(name, command)
```

Sometimes, though, we'll want our button to stand out; maybe instead of black text on a gray background, we want red text on a green background. This necessitates giving even more arguments to the `AButton constructor`:

```
my_button = AButton(name, command, foreground_color,
                    background_color)
```

Even this isn't enough. We may want to have buttons with thicker borders, or buttons with specific heights or widths, or buttons that contain a small picture instead of text. We could easily require an `AButton` command with 20 different arguments, with the end result being that the `AButton` command would be practically unusable and wouldn't give us the control we want.

Tkinter solves this problem by specifying almost all properties of widgets as *named attributes*, values that can optionally be given by name when the widget is created and that can be accessed or modified by that same name later in the life of the widget. This works well with Python's named parameter passing, making it easy to create widgets with the desired attributes. For example, the name of the attribute that defines the string displayed in a button is `text`. The Python command to create a button that displays the string `"Hello!"` is

```
my_button = Button(text="Hello!")
```

Most attributes of a widget have a default value, which is used when you don't supply a value for that attribute. Generally speaking, these defaults make sense for the common case, and most of the time most named attributes can be ignored. For example, the named attribute that controls the color of the text in a button is called `foreground`, and its default value is the string `"black"`, which causes the button text to display as black. To override this default, give a specific value for the `foreground` attribute:

```
my_button = Button(text="Hello!", foreground="red")")
```

Named attributes are used extensively throughout the Tkinter widget set, and many attributes can be used with almost all widgets. The use of named attributes greatly simplifies the process of GUI programming and makes code more readable.

16.3.3 Geometry management and widget placement

The final basic aspect of Tkinter that you need to understand is the idea of geometry management, meaning how widgets are placed on the screen. It isn't obvious in the previous example, but typing in a line of Python code like so

```
my_button = Button(text="Alright!")
```

isn't enough to display the button onscreen. Tk doesn't know where you want the button to show up, and until it's told the desired position, it will keep the button hidden. Deciding where to display the button onscreen is a function of the window hierarchy and associated Tk geometry managers.

To understand the idea of the Tk window hierarchy, you need to know about two special Tk widget classes, called `Toplevel` and `Frame`. Both `Toplevel` widgets and `Frame` widgets may contain other Tk widgets (including other frames) and are the basic building blocks for constructing complex GUIs. A `Frame` is a container for other widgets and can be either the main window of an application or contained in another frame. Nesting frames within frames can be useful in laying out and grouping widgets.

Tk uses a `Toplevel` widget to represent a complete window in a GUI, complete with title bar, close and zooming buttons, and so forth. A `Toplevel` widget is useful for custom dialog boxes and other situations where you want a window that's independent from the main `Frame`.

The subwidgets contained in any particular frame are arranged for display according to one of Tkinter's three built-in geometry managers. These managers permit you to specify the arrangement of the subwidgets in various ways, ranging from giving exact coordinates for each widget within a window to giving only relative placement, leaving the precise sizing of each widget to the geometry manager.

For the purposes of talking about Tkinter in this chapter, we'll refer to only one geometry manager: the grid manager, which is the most powerful. `grid` works by placing all widgets in an implicit grid, similar to a spreadsheet layout. If you start using Tkinter, you'll want to learn the pack and place geometry managers as well, which you can use to specify that widgets should be placed relative to one another (pack), or in absolute locations in a window (place).

16.4 A simple Tkinter application

Let's start with an example that introduces all of the basic Tkinter concepts: window hierarchies, geometry management, Tkinter attributes, and a couple of the most basic widgets. The example is a simple one. When run, it produces a window that resembles figure 16.2. It may look different on your machine, because Tk provides a native look and feel for whatever operating system it's running on. This example was produced under the Windows XP operating system.



Figure 16.2 A simple application

Clicking the Increment button adds 1 to the number shown in the Count field, and clicking the Quit button quits the application.

Listing 16.1 contains the code to do this.

Listing 16.1 File tk_count.py

```
from tkinter import *
main_window = Tk()                                         ❶
count_label = Label(main_window, text="Count: 0")              ❷      ❹
```

```
count_label.grid(row=0, column=1)                     ❸
count_value = 0          Starts counter value at 0

def increment_count():                   Called when Increment button is clicked
    global count_value, count_label
    count_value = count_value + 1                       Increments counter
    count_label.configure(text='Count: ' + str(count_value))
incr_button = Button(main_window, text="Increment",        ❷    ❹
                     command=increment_count)
incr_button.grid(row=0, column=0)                      ❸
quit_button = Button(main_window, text="Quit",          ❷     ❹
                    command=main_window.destroy)
quit_button.grid(row=1, column=0)                      ❸
mainloop()                                              Reenters Tk event loop
```

The example shows the basic principles of Tkinter-based programming:

- *Widget creation, accomplished here by the* `Label` *and* `Button` *commands* ❷—Tkinter lets you create many different types of widgets, such as lists, scroll bars, dialog boxes, radio and check buttons, and so on.
- *Widget placement, accomplished in this case by the* `grid` *command* ❸—Tk provides a great deal of control over how widgets are placed and sized. `grid` will be discussed in more detail in a later section.
- *The use of widget attributes, to set and modify the appearance and behavior of widgets* ❹—The widget attributes used in the example are the `text` attribute, which controls the text displayed by a widget, and the `command` attribute, which sets the function the widget will execute when it's clicked. You can set widget attributes when a widget is first created and change them after a widget has been created by using the `configure` widget method.
- *A basic window hierarchy created by the program* ❶—The `main_window` is the top-level window widget. It, in turn, contains the `count_label`, `incr_button`, and `quit_button` widgets.

16.5 Creating widgets

You create widgets in Python by instantiating an object of that widget's class, using the name of the type of widget being created. `Button` and `Label` widgets were created in the previous example, but you can also create `Menu`, `Scrollbar`, `Listbox`, `Text`, and many other types.

The widget-creation commands all follow the same general form. They all have one mandatory argument, the parent window (or parent widget), followed by zero or more optional named widget attributes, which determine the precise appearance and behavior of the new widget. Each creation command returns the new widget as a result. You'll usually want to store this new widget somewhere so that you can modify it later if necessary. A line in a Python program that creates a widget usually looks something like this:

```
new_widget = WidgetCreationCommand(parent, attribute1=value1,
                                           attribute2=value2, . . .)
```

The parent window of a widget called w is the window (or widget) that contains w.[1] It's important to define a parent for several reasons. First, widgets are always displayed inside and relative to their parent window. Second, Tk provides a powerful event mechanism (which we don't have space to discuss), and a widget may pass events to its parent if it can't handle them itself. Finally, Tk can have widgets that act as windows, in that they themselves are the parent window for (and contain) other widgets, and the widget-creation commands need to be able to set up this sort of relationship.

All the other optional arguments in a widget-creation command define widget attributes and control different aspects of the widget. Some widget attributes are common to several different widget types (for example, the text attribute applies to all types of simple widgets that can display a label or a line of text of some sort), whereas other attributes are unique to certain widgets. One characteristic that makes Tk special, compared to other GUI packages, is that it gives you a great deal of control over your widgets. There are many attributes for each widget. To give you an idea of this, here's a program that uses some of the attributes that control the appearance of widgets. Figure 16.3 is the resulting window (in black and white, unfortunately):

```
from tkinter import *
main_window = Tk()
label = Label(main_window, text="Hello", background='white',
              foreground='red', font='Times 20',
              relief='groove', borderwidth=3)
label.grid(row=0, column=0)
mainloop()
```



Figure 16.3 A widget window

Most attributes of widgets can be set at creation in this way.

16.6 Widget placement

Creating a widget doesn't automatically draw it on the screen. Before this can be done, Tk needs to know where the widget should be drawn. The grid command was used in the previous example and will be discussed in detail.

Tk is more sophisticated in the way it handles widget placement than most GUI packages. Under Windows, the standard way of setting the locations of widgets is to specify an absolute position in their parent window. This can also be done in Tkinter (using the place rather than the grid command) but usually isn't, because this technique isn't very flexible. For instance, if you set up a window for use on a monitor that has 640×480 resolution, and a user uses it on a monitor that has 1600×1200 resolution, the window uses only a small amount of the available screen space and can't be resized (unless you write the code to resize the window). This is a common problem with many programs.

---

1 This is an oversimplification. The parent window of a widget is generally the widget that contains that widget. But this isn't strictly necessary.

Instead, Tkinter usually makes use of the notion of *relative placement*, where widgets are placed in such a manner that their positions relative to one another are maintained no matter what size the enclosing window happens to be. This can get complex. For example, you can specify that widget A should be to the left of widget B, and that when the enclosing window is resized, widget A should grow to take advantage of the extra space, but widget B shouldn't. We won't get into all of the possibilities but will attempt to present enough of the features of Tk widget placement to illustrate the ease and power of the methods it uses.

The `grid` command places widgets in a window by considering a window as an infinite grid of cells. You place a widget into this grid by specifying `row` and `column` arguments to the `grid` command, which tell it in which cell to place the widget. The rows and columns will expand as needed to display the widgets they contain, and any rows or columns that don't display any widgets aren't displayed.

As a simple example, we'll put two buttons in the corners of a 2?×?2 grid:

```
from tkinter import *

win = Tk()
button1 = Button(win, text="one")
button2 = Button(win, text="two")
button1.grid(row=0, column=0)
button2.grid(row=1, column=1)
mainloop()
```

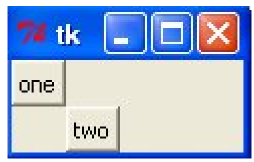


Figure 16.4 A two-button window

When run, this program produces a window that looks like figure 16.4.

The cells of the grid are automatically sized large enough to display what they contain, although you can override this and place constraints on the maximum sizes of the cells.

This makes it easy to set up a text window with scrollbars and, with the proper placement (see figure 16.5), treat the window as a 2×?2 grid into which the `Text` and `Scrollbar` widgets will be placed.

The program to do this is shown in listing 16.2.

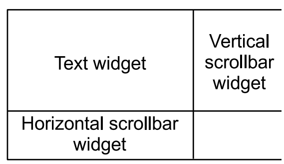


Figure 16.5 Grid usage

Listing 16.2 File tk_grid.py

```
from tkinter import *
main = Tk()
main.columnconfigure(0, weight=1)
main.rowconfigure(0, weight=1)
text = Text(main)
text.grid(row=0, column=0, sticky='nesw')
vertical_scroller = Scrollbar(main, orient='vertical')
vertical_scroller.grid(row=0, column=1, sticky='ns')
horizontal_scroller = Scrollbar(main, orient='horizontal')
horizontal_scroller.grid(row=1, column=0, sticky='ew')
mainloop()
```

The commands in the code ensure that any extra space given to the grid as a result of resizing the top-level window is allocated to column 0, row 0—that is, to the Text widget. The resulting window is shown in figure 16.6.



Figure 16.6 A text window

In addition to the three `grid` method invocations that place the text box and two scrollbars in their appropriate cells, this code reveals new aspects of Tk, which build on the fundamentals of Tk to provide further control over the user interface:

- The `orient` attributes of the `Scrollbar` widgets control whether a scrollbar scrolls vertically or horizontally.
- The `sticky` attributes of all three widgets control how they're placed in their cells. For example, the `Text` widget has a sticky value of `'nesw'`, which means that its north (top) side should stick to the north side of the cell it's in, its east (right) side should stick to the east side of its containing cell, and similarly for the south and west sides. The `Text` widget should completely fill its cell, which means that if its cell grows, the text widget should also grow and automatically reformat the text it contains to take advantage of the extra space. It's then fairly easy to guess that the sticky value of `'ns'` for the vertical scrollbar means it should always stretch in the vertical direction to fill its cell (and always be the same height as the `Text` widget), and analogously for `'ew'` and the horizontal scrollbar.
- The `columnconfigure` command specifies that if the window containing the entire grid of widgets is expanded in the horizontal direction, all the extra space resulting from the resizing should be allocated to column 0, the column containing the `Text` widget. `rowconfigure` specifies similarly in the vertical direction. Together, `columnconfigure` and `rowconfigure` ensure that if the top level is resized to be larger, then the extra space resulting from that resizing should be given to the `Text` widget, which is generally the desired behavior.

That's a lot of GUI detail specification in a small amount of space. But that's exactly the point of Tk—to enable you to rapidly specify your GUI, right down to the details. The fact that settings are accomplished through easy-to-remember keywords, rather than a multitude of binary flags, doesn't hurt either.

16.7 Using classes to manage Tkinter applications

One problem with creating Tkinter applications as we have so far is that they quickly become hard to read and maintain as you add more widgets and code. Using the OOP principles introduced in the previous chapter and making an application class that

inherits from `Frame` can make your code much more organized and easy to read and maintain. Listing 16.3 shows the previous counter application, refactored as a class.

Listing 16.3 File tk_count_oop.py

```
from tkinter import *
class Application(Frame):
    def __init__(self, master=None):
        Frame.__init__(self, master)
        self.grid()
        self.create_widgets()
        self.count_value = 0

    def create_widgets(self):
        self.count_label = Label(self, text="Count: 0")
        self.count_label.grid(row=0, column=1)
        self.incr_button = Button(self, text="Increment",
                                  command=self.increment_count)
        self.incr_button.grid(row=0, column=0)
        self.quit_button = Button(self, text="Quit",
                               command=self.master.destroy)
        self.quit_button.grid(row=1, column=0)

    def increment_count(self):
        self.count_value += 1
        self.count_label.configure(text='Count: ' + str(self.count_value))

app = Application()
app.mainloop()
```

Annotations: Creates application class from Tkinter's Frame class. (class Application); Create instance of app (app = Application()); Runs app (app.mainloop()).

This code is no longer than the previous version, but it's much better organized and easier to read. Basic initialization, widget setup, and counter incrementing are now much easier to pick out; and because `increment_counter` is now an instance variable, we no longer need to make it global. Even though you almost certainly won't want more than one instance of `Application` at a time, creating the class is worth it because of the improved organization (and readability and maintainability) it gives your code. Use this pattern for your Tkinter applications—it will more than pay off.

16.8 What else can Tkinter do?

The previous examples don't come close to exploring the capabilities of Tkinter, but they should give you some idea of what it feels like. There's no way in the space of one chapter to illustrate by example all the facilities Tk provides. The next few sections will cover the remainder of Tkinter's abilities on a much higher level. If you're familiar with GUI programming, you'll be at home with most of what is discussed.

If you aren't familiar with GUI programming, the examples so far are a good starting point for learning by experimentation. They should give you grounding in Tk's basic philosophy and concepts. You can find further detailed information linked on the Tkinter page on the Python website. Also be sure to check out John Grayson's comprehensive book *Python and Tkinter Programming* (Manning, 2000).

16.8.1 Event handling

Event handling—how a GUI library handles user actions such as mouse movements or clicks or keypresses—is a critical part of GUI programming. An awkward event-handling scheme can make your GUI development a real nightmare, whereas a good event-handling mechanism can make your task far more pleasant. Tk event handling, although not perfect, is up near the top. One of the big questions with event handling is *event direction*—if you press a key on the keyboard, which text entry box or text widget (or other widget) of the many in your interface is that directed to? Many GUI libraries require a complete specification of how events are to be directed; you must say, "If this widget can't handle a mouse click event, it should pass that event over to this other widget, and so on." You can do this in Tk, but you aren't required to. Tk uses the hierarchical structure of your user interface, together with various commonsense rules, to try to automatically direct events to the appropriate widget. Most of the time, it gets it right.

Tk also provides a rich set of events to choose from. For example, look at the problem of changing the mouse cursor to a paintbrush, as the mouse enters the main paint window of a painting program. One way to do this would be to continually (explicitly) monitor the position of the mouse relative to the main paint window and manually adjust the cursor as necessary. This would be necessary if only basic mouse-related events, such as movements, were reported by and to Tk. But Tk provides a higher-level event for all windows, subwindows, and widgets, called the `<Enter>` event. An `<Enter>` event for a window or widget is generated every time the mouse cursor enters that window or widget (and a corresponding `<Leave>` event is generated when the mouse leaves). A built-in `bind` command makes it easy to bind `<Enter>` events for a particular window, such as a painting window, to specific functions, such as a function to change the mouse cursor to a paintbrush.

If the built-in set of Tk events isn't enough, you can define or generate your own *virtual events*. These are events defined by you, just as you can define application-specific functions. For example, you might wish to define the virtual event called `<<Copy>>` and set it to be equivalent to the keyboard event generated by pressing Ctrl-C. If you ever migrated your program to a platform or language (such as Chinese) where Ctrl-C wasn't an appropriate key binding for a `<<Copy>>` event, you could redefine `<<Copy>>` in terms of another keystroke—but all references to `<<Copy>>` in your code would remain valid, without needing any changes. Generating virtual events is also a convenient way to distribute events widely throughout your application without worrying about plumbing details. For example, you could define a button called MyButton, such that clicking this button would generate a `<<MyButtonPressed>>` event. Any other widgets could then be instructed to listen for `<<MyButtonPressed>>` events without referencing the original button—indeed, without knowing or caring that a button was the source of the event.

16.8.2 Canvas and text widgets

Two widgets in the Tk widget set deserve special mention, because they provide abilities reflecting literally years of implementation effort. These are the `Canvas` and `Text` widgets, which respectively provide high-level manipulation capabilities for object-oriented graphics and for text. Both widget types are an order of magnitude more advanced than analogs found in most other GUI libraries.

The `Text` widget supports all the basic functionality necessary for a basic WYSIWYG word processor, including font families, styles, and sizes; a rich set of default key bindings; automatic controllable line wrapping; settable tabs; the ability to embed images or other widgets onto the text drawing surface; and so forth. In addition, you can tag text within the widget with any number of user-chosen strings and then manipulate the text via those tags. You could, for example, define a tag called `bold` to be applied to all text in your widget that should be displayed in boldface. A single line of code would then let you change the display style for all bold text from Helvetica 10 pt. bold to Times Roman 14 pt. bold, or, if you prefer, to red text on a green background with a 2-pixel-wide raised border. You can easily define sections of text that highlight as the mouse cursor passes over them and that cause some action when they're clicked, mimicking the effect of buttons or hypertext. Many other abilities also come with the `Text` widget.

The `Canvas` widget is similar, particularly with respect to the use of tags. You can associate arbitrary tags with an object and manipulate as a whole all objects on the canvas that have a given tag. For example, it's an easy matter to define a complex shape by drawing a series of lines, curves, and other shapes, all with the same programmer-defined tag, and then to move that shape around the canvas as a unit, by instructing the `Canvas` widget to move all simple shapes having the given tag. It's also easy to use shapes as buttons, to set various aspects of their appearance (such as foreground and background colors and line width), to define layering of shapes atop one another, and to perform many other advanced tasks that would take months of effort if undertaken from scratch.

16.9 Alternatives to Tkinter

Tkinter may not satisfy your needs. It isn't particularly fast and isn't a good candidate for games or for image-manipulation programs. Its high-level approach means that particularly unusual or specialized user interfaces may be difficult to implement. Or, you may not have the time to learn Tkinter. Fortunately, alternatives are available.

Three cross-platform windowing/GUI libraries stand out, being available for at least Windows, Mac OS X, and Linux/UNIX. The first is the Qt package, which forms the basis for the well-done KDE desktop environment effort, a large project geared toward producing an integrated and comprehensive desktop environment for Linux/UNIX and compatible operating systems. QT is a rich and powerful GUI framework

and also has QT Designer, a capable GUI builder. You can find out more about Qt at http://qt.nokia.com. Also, www.kde.org will give you a chance to browse screenshots of many applications built using KDE.

The next cross-platform GUI option is GTK. This is the GIMP Toolkit and is the basis for the GNOME desktop environment project, which is comparable to KDE. GTK is similar to Qt in scope and capability, and it has also been ported from Linux/UNIX to work on Windows and Mac OS X. Another interesting option with GTK is Glade, a graphical tool to build GUI interfaces for GTK. Glade saves its files in XML format, which can be used directly with a library called libglade or used to generate the appropriate code. A good starting point for finding out about GTK is the GTK website, at www.gtk.org.

Finally, there is the wxPython toolkit, based on the wxWidgets framework. wxWidgets was created to be portable across Windows and UNIX platforms and has since been extended to Mac OS X and other platforms. wxPython is a strong framework, offering a native look and feel on different platforms, and is widely used. Visit www.wxpython.org to find out more about wxPython.

One thing to keep in mind about all of these GUI options is that they require you to install both the GUI libraries themselves *and* the Python libraries to use them. This means more work on the user end of things and can lead to issues with making sure the right versions of all the dependencies are installed. On the other hand, developers in many situations believe the tradeoff is more than worth it.

Many other GUI libraries are available. Good descriptions and evaluations are available on the Python wiki's Gui Programming page, which you can find by visiting the main python.org website and searching for "gui programming."

16.10 Summary

Python ships with a comprehensive and well-thought-out interface (a Python module) called Tkinter, which allows access to the freely available and powerful Tk user interface library. Tkinter is a handy framework for a scripting-style language, in that it allows rapid interface development, interactive execution, and runtime control over all aspects of the interface, with powerful abilities. It suffers some of the same drawbacks of scripting languages, particularly execution overhead that may be too high for graphics-intense applications, and inflexibility in the UI model it uses.

There are many alternatives. If you're programming only for the Windows environment, you can take advantage of your MS UI library knowledge through direct calls to the Microsoft APIs. Qt, GTK, and wxPython are all excellent cross-platform choices.

Part 3

Advanced language features

The previous chapters have been a survey of the basic features of Python: what 80 percent of programmers will use 80 percent of the time. What follows is a brief look at some more advanced features, which you may not use every day but which are handy when you need them.

17 Regular expressions

This chapter covers

- Understanding regular expressions
- Creating regular expressions with special characters
- Using raw strings in regular expressions
- Extracting matched text from strings
- Substituting text with regular expressions

In some sense, we shouldn't discuss regular expressions in this book at all. They're implemented by a single Python module and are advanced enough that they don't even come as part of the standard library in languages like C or Java. But if you're using Python, you're probably doing text parsing; and if you're doing that, then regular expressions are too useful to be ignored. If you use Perl, Tcl, or UNIX, you may be familiar with regular expressions; if not, this chapter will go into them in some detail.

17.1 What is a regular expression?

A *regular expression* (RE) is a way of recognizing and often extracting data from certain patterns of text. A regular expression that recognizes a piece of text or a string is said to *match* that text or string. An RE is defined by a string in which certain of

the characters (the so-called *metacharacters*) can have a special meaning, which enables a single RE to match many different specific strings..

It's easier to understand this through example than through explanation. Here's a program using a regular expression, which counts how many lines in a text file contain the word *hello*. A line that contains *hello* more than once will be counted only once:

```
import re
regexp = re.compile("hello")
count = 0
file = open("textfile", 'r')
for line in file.readlines():
    if regexp.search(line):
        count = count + 1
file.close()
print(count)
```

The program starts by importing the Python regular expression module, called `re`. It then takes the text string `"hello"` as a *textual regular expression* and compiles it into a *compiled regular expression*, using the `re.compile` function. This isn't strictly necessary, but compiled regular expressions can significantly increase a program's speed, so they're almost always used in programs that process large amounts of text.

What can the regular expression compiled from `"hello"` be used for? You can use it to recognize other instances of the word `"hello"` within another string; in other words, you can use it to determine whether another string contains `"hello"` as a substring. This is accomplished by the `search` method, which returns `None` if the regular expression isn't found in the string argument; Python interprets `None` as `false` in a Boolean context. If the regular expression is found in the string, than Python returns a special object that you can use to determine various things about the match (such as where in the string it occurred). We'll discuss this later.

17.2 Regular expressions with special characters

The previous example has a small flaw—it counts how many lines contain `"hello"` but ignores lines that contain `"Hello"` because it doesn't take capitalization into account.

One way to solve this would be to use two regular expressions, one for `"hello"` and one for `"Hello"`, and test each of these REs against every line. A better way is to use the more advanced features of regular expressions. For the second line in the program, substitute

```
regexp = re.compile("hello|Hello")
```

This regular expression uses the vertical bar special character `|`. A special character is a character in a regular expression that isn't interpreted as itself—it has some special meaning. `|` means *or*, so the regular expression matches `"hello"` *or* `"Hello"`.

Another way of doing this is to use

```
regexp = re.compile("(h|H)ello")
```

In addition to using `|`, this regular expression uses the *parentheses* special characters to group things, which in this case means that the `|` only chooses between a small or capital *H*. The resulting regular expression matches either an *h* or an *H*, followed by *ello*.

Another way of performing the match is

```
regexp = re.compile("[hH]ello")
```

The special characters `[` and `]` take a string of characters between them and match any single character in that string. There's a special shorthand to denote ranges of characters in `[` and `]`; `[a-z]` will match a single character between *a* and *z*, `[0-9A-Z]` will match any digit or any uppercase character, and so forth. Sometimes you may want to include a real hyphen in the `[]`, in which case you should put it as the first character to avoid defining a range; `[-012]` will match a hyphen, or a *0* or a *1* or a *2*, and nothing else.

Quite a few special characters are available in Python regular expressions, and describing all of the subtleties of using them in regular expressions is beyond the scope of this book. A complete list of the special characters available in Python regular expressions, as well as descriptions of what they mean, is given in the appendix at the end of this book. For the remainder of this chapter, we'll describe the special characters we use as they appear.

17.3 Regular expressions and raw strings

The functions that compile REs, or search for matches to REs, understand that certain character sequences in strings have special meanings in the context of regular expressions. For example, RE functions understand that `\n` represents a newline character. But if you use normal Python strings as regular expressions, the RE functions will typically never see such special sequences, because many of these sequences also possess a special meaning in normal strings. `\n`, for example, also means newline in the context of a normal Python string, and Python will automatically replace the string sequence `\n` with a newline character before the `RE` function ever sees that sequence. The `RE` function, as a result, will compile strings with embedded newline characters—not with embedded `-\n` sequences.

In the case of`\n`, this makes no difference because `RE` functions interpret a newline character as exactly that and do the expected thing—they attempt to match it with another newline character in the text being searched. Let's look at another special sequence, `\\`, which represents a *single* backslash to REs. Assume that we wish to search some text for an occurrence of the string `"\ten"`. Because we know that we have to represent a backslash as a double backslash, we might try

```
regexp = re.compile("\\ten")
```

This will compile without complaining, but it's wrong. The problem is that `\\` also means a single backslash in Python strings. Before `re.compile` is invoked, Python interprets the string we typed as meaning `\ten`, which is what is passed to `re.compile`. In the context of regular expressions, `\t` means *tab*, so our compiled regular expression searches for a tab character followed by the two characters *en*.

To fix this while using regular Python strings, we need four backslashes. Python interprets the first two backslashes as a special sequence representing a single backslash, and

likewise for the second pair of backslashes, resulting in two *actual* backslashes in the Python string. That string is then passed in to `re.compile`, which interprets the two actual backslashes as an RE special sequence representing a single backslash. Our code looks like this:

```
regexp = re.compile("\\\\ten")
```

That seems confusing, and it's why Python has a way of defining strings, called *raw strings*.

17.3.1 Raw strings to the rescue

A raw string looks similar to a normal string, except that it has a leading *r* character immediately preceding the initial quotation mark of the string. Here are some raw strings:

```
r"Hello"
r"""\tTo be\n\tor not to be"""
r'Goodbye'
r'''12345'''
```

As you can see, you can use raw strings with either the single or double quotation marks and with the regular or triple-quoting convention. You can also use a leading *R* instead of *r* if you wish. No matter how you do it, raw string notation can be taken as an instruction to Python saying, *Don't process special sequences in this string*. In the previous examples, all the raw strings are equivalent to their normal string counterparts except the second example, in which the `\t` and `\n` sequences aren't interpreted as tabs or newlines but are left as two-string character sequences beginning with a backslash.

Raw strings aren't a different type of string. They're a different way of *defining* strings. It's easy to see what's happening by running a few examples interactively:

```
>>> r"Hello" == "Hello"
True
>>> r"\the" == "\\the"
True
>>> r"\the" == "\the"
False
>>> print(r"\the")
\the
>>> print("\the")
        he
```

Using raw strings with regular expressions means you don't need to worry about any funny interactions between string special sequences and regular expression special sequences. You use the regular expression special sequences. The previous RE example then becomes

```
regexp = re.compile(r"\\ten")
```

which works as expected. The compiled RE looks for a single backslash followed by the letters *ten*.

You should get into the habit of using raw strings whenever defining REs, and we'll do so for the remainder of this chapter.

17.4 Extracting matched text from strings

One of the most common uses of regular expressions is to perform simple pattern-based parsing on text. This is something you should know how to do, and it's also a good way to learn more regular expression special characters.

Assume, for example, that we have a list of people and phone numbers in a text file. Each line of the file will look like this

```
surname, firstname middlename: phonenumber
```

with a surname, followed by a comma and space, followed by a first name, followed by a space, followed by a middle name, followed by colon and a space, followed by a phone number.

But to make things complicated, the middle name may or may not exist, and the phone number may or may not have an area code. It might be 800-123-4567, or it might be 123-4567. You *could* write code to explicitly parse data out from such a line, but it would be a tedious and error-prone job. Regular expressions provide a simpler answer.

We'll start by coming up with a regular expression that will match lines of the given form. The next few paragraphs will throw quite a few special characters at you. Don't worry if you don't get them all on the first read—as long as you understand the gist of things, that's all right.

For simplicity's sake, let's assume for right now that that first names, surnames, and middle names consist of letters and possibly a hyphen. We can use the `[]` special characters defined in the previous section to define a pattern that defines only name characters:

```
[-a-zA-z]
```

This pattern will match a single hyphen, or a single lowercase letter, or a single uppercase letter.

To match a full name (like McDonald), we need to repeat this pattern. The `+` metacharacter repeats whatever comes before it one or more times as necessary to match the string being processed. So, the pattern

```
[-a-zA-Z]+
```

will match a single name, like Kenneth or McDonald or Perkin-Elmer. It will also match some strings that aren't names, like — or -a-b-c-, but that's all right for our purposes.

Now, what about the phone number? The special sequence `\d` matches any digit, and a hyphen outside of `[]` is a normal hyphen. A good pattern to match the phone number is

```
\d\d\d-\d\d\d-\d\d\d\d
```

That's three digits, followed by a hyphen, followed by three digits, followed by a hyphen, followed by four digits. This will match only phone numbers with an area code, and our list may contain numbers that don't have one. The best solution is to enclose the area code part of the pattern in `()`, group it, and then follow that group

with a ? special character, which says that the thing coming immediately before the ? is optional:

```
(\d\d\d-)?\d\d\d-\d\d\d\d
```

This pattern will match a phone number that may or may not contain an area code. We can use the same sort of trick to account for the fact that some of the people in our list have their middle name included, and some don't. (To do this, make the middle name optional using grouping and the ? special character.)

Commas, colons, and spaces don't have any special meaning in regular expressions (they mean themselves). Putting everything together, we come up with a pattern that looks like this:

```
[-a-zA-Z]+, [-a-zA-Z]+( [-a-zA-Z]+)?: (\d\d\d-)?\d\d\d-\d\d\d\d
```

A real pattern would probably be a bit more complex, because we wouldn't assume that there is exactly one space after the comma, exactly one space after the first and middle names, and exactly one space after the colon. But that's easy to add later.

The problem is that, whereas the above pattern will let us check to see if a line has the anticipated format, we can't extract any data yet. All we can do is write a program like this:

```
import re
regexp = re.compile(r"[-a-zA-Z]+,"
                    r" [-a-zA-Z]+"
                    r"( [-a-zA-Z]+)?"
                    r": (\d\d\d-)?\d\d\d-\d\d\d\d"
                   )
file = open("textfile", 'r')
for line in file.readlines():
    if regexp.search(line):
        print("Yeah, I found a line with a name and number. So what?")
file.close()
```

Last name and comma (line `r"[-a-zA-Z]+,"`)

First name (line `r" [-a-zA-Z]+"`)

Optional middle name (line `r"( [-a-zA-Z]+)?"`)

Colon and phone number (line `r": (\d\d\d-)?\d\d\d-\d\d\d\d"`)

Notice that we have split up our regular expression pattern using the fact that Python will implicitly concatenate any set of strings separated by whitespace. As your pattern grows, this can be a great aid in keeping it maintainable and understandable. It also solves the problem with the line length possibly increasing beyond the right edge of the screen.

Fortunately, you can use regular expressions to extract data from patterns, as well as to check to see if the patterns exist. The first part of doing this is to group each subpattern corresponding to a piece of data you wish to extract using the () special characters and then give each subpattern a unique name with the special sequence ?P<name>, like this:

```
(?P<last>[-a-zA-Z]+), (?P<first>[-a-zA-Z]+)( (?P<middle>([-a-zA-Z]+)))?:
(?P<phone>(\d\d\d-)?\d\d\d-\d\d\d\d)
```

(Please note you should enter these lines as a single line with no line breaks. Due to space constraints, we couldn't represent it here in that manner.)

There's an obvious point of confusion here: The question marks in `?P<...>` and the question mark special characters that say the middle name and area code are optional have nothing to do with one another. It's an unfortunate semicoincidence that they happen to be the same character.

Now that we have named the elements of the pattern, we can extract them as matches are made, by using the `group` method. This is possible because when the `search` function returns a successful match, it doesn't just return a truth value; it returns a data structure that records what was matched. We can write a simple program to extract names and phone numbers from our list and print them right out again as follows:

```
import re
regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"                    <-- Last name and comma
                    r" (?P<first>[-a-zA-Z]+)"                   <-- First name
                    r"( (?P<middle>([-a-zA-Z]+)))?"             <-- Optional middle name
                    r": (?P<phone>(\d\d\d-)?\d\d\d-\d\d\d\d)"   <-- Colon and phone number
                   )
file = open("textfile", 'r')
for line in file.readlines():
    result = regexp.search(line)
    if result == None:
        print("Oops, I don't think this is a record")
    else:
        lastname = result.group('last')
        firstname = result.group('first')
        middlename = result.group('middle')
        if middlename == None:
            middlename = ""
        phonenumber = result.group('phone')
    print('Name:', firstname, middlename, lastname,' Number:', phonenumber)
file.close()
```

There are some points of interest here:

- We can find out whether a match succeeded by checking the value returned by `search`. If the value is `None`, the match failed; otherwise, the match succeeded, and we can extract information from the object returned by `search`.
- `group` is used to extract whatever data matched with our named subpatterns. We pass in the name of the subpattern we're interested in.
- Because the middle subpattern is optional, we can't count on it having a value, even if the match as a whole is successful. If the match succeeds, but the match for the middle name doesn't, then using `group` to access the data associated with the middle subpattern will return the value `None`.
- Part of the phone number is optional, but part isn't. If the match succeeds, the phone subpattern must have some associated text, so we don't have to worry about it having a value of `None`.

17.5 Substituting text with regular expressions

In addition to extracting strings from text, you can use Python's regular expression module to find strings in text and substitute other strings in place of those that were found. You accomplish this using the regular substitution method `sub`. The following example replaces instances of `"the the"` (presumably a typo) with single instances of `"the"`:

```
>>> import re
>>> string = "If the the problem is textual, use the the re module"
>>> pattern = r"the the"
>>> regexp = re.compile(pattern)
>>> regexp.sub("the", string)
'If the problem is textual, use the re module'
```

The `sub` method uses the invoking regular expression (`regexp`, in this case) to scan its second argument (`string`, in the example) and produces a new string by replacing all matching substrings with the value of the first argument (`"the"`, in this example).

But what if you want to replace the matched substrings with new ones that reflect the value of those that matched? This is where the elegance of Python comes into play. The first argument to `sub`—the replacement substring, `"the"` in the example—doesn't have to be a string at all. Instead, it can be a function, and if it's a function, Python calls it with the current match object and lets that function compute and return a replacement string.

To see this in action, we'll build an example that will take a string containing integer values (no decimal point or decimal part) and return a string with the same numerical values but as floating numbers (with a trailing decimal point and zero):

```
>>> import re
>>> int_string = "1 2 3 4 5"
>>> def int_match_to_float(match_obj):
...     return(match_obj.group('num') + ".0")
...
>>> pattern = r"(?P<num>[0-9]+)"
>>> regexp = re.compile(pattern)
>>> regexp.sub(int_match_to_float, int_string)
'1.0 2.0 3.0 4.0 5.0'
```

In this case, the pattern looks for a number consisting of one or more digits (the `[0-9]+` part). But it's also given a name (the `?P<num>...` part) so that the replacement string function can extract any matched substring by referring to that name. The `sub` method then scans down the argument string `"1 2 3 4 5"`, looking for anything that matches `[0-9]+`. When `sub` finds a substring that matches, it makes a match object defining exactly which substring has matched the pattern, and it calls the `int_match_to_float` function with that match object as the sole argument. `int_match_to_float` uses `group` to extract the matching substring from the match object (by referring to the group name `num`) and produces a new string by concatenating the matched substring with a `".0"`. `sub` returns the new string and incorporates it

as a substring into the overall result. Finally, `sub` starts scanning again right after the place where it found the last matching substring, and it keeps going like that until it can't find any more matching substrings.

17.6 Summary

I wish I could say I have provided a reasonably comprehensive overview of the regular expression abilities of Python. I haven't—not by a long shot. I've attempted to give a good introduction to the topic and to give a reasonable description of the most important of the regular expression facilities in Python. I've skipped many of the regular expression special characters, but you can find a complete list in the Python documentation. I've talked about the `search` and `sub` methods but omitted many other methods that can be used to split strings, extract more information from match objects, look for the positions of substrings in the main argument string, and precisely control the iteration of a regular expression search over an argument string. I've mentioned the `\d` special sequence, which can be used to indicate a digit character, but there are many other special sequences—they're listed in the documentation. And I've failed to mention regular expression flags, which you can use to control some of the more esoteric aspects of how extremely sophisticated matches are carried out.

If you do a lot of text processing and searching, I'd strongly suggest learning as much about regular expressions as possible. Start by becoming familiar with the features described in this chapter, and delve into the additional features described in the documentation as you become more comfortable with regular expressions.

In addition to the section on regular expressions in the *Python Library Reference Manual*, you can find an excellent tutorial written by Andrew Kuchling at the Python website.

18 Packages

This chapter covers

- Defining a package
- Creating a simple package
- Exploring a concrete example
- Using the `__all__` attribute
- Using packages properly

Modules make reusing small chunks of code easy. The problem comes when the project grows and the code you want to reload outgrows, either physically or logically, what would fit into a single file. If having one giant module file is an unsatisfactory solution, having a host of little unconnected modules isn't much better. The answer to this problem is to combine related modules into a package.

18.1 What is a package?

A *module* is a file containing code. A module defines a group of usually related Python functions or other objects. The name of the module is derived from the name of the file.

When you understand modules, packages are easy, because a package is a directory containing code and possibly further subdirectories. A package contains a

group of usually related code files (modules). The name of the package is derived from the name of the main package directory.

Packages are a natural extension of the module concept and are designed to handle very large projects. Just as modules group related functions, classes, and variables, packages group related modules.

18.2 A first example

To see how this might work in practice, let's sketch a design layout for a type of project that by nature is very large—a generalized mathematics package, along the lines of Mathematica, Maple, or MATLAB. Maple, for example, consists of thousands of files, and some sort of hierarchical structure is vital to keeping such a project ordered. We'll call our project as a whole `mathproj`.

We can organize such a project in many ways, but a reasonable design splits the project into two parts: `ui`, consisting of the user interface elements, and `comp`, the computational elements. Within `comp`, it may make sense to further segment the computational aspect into `symbolic` (real and complex symbolic computation, such as high school algebra) and `numeric` (real and complex numerical computation, such as numerical integration). It may then make sense to have a constants.py file in both the `symbolic` and `numeric` parts of the project.

The constants.py file in the numeric part of the project defines pi as

```
pi = 3.141592
```

whereas the constants.py file in the symbolic part of the project defines pi as

```
class PiClass:
    def __str__(self):
        return "PI"
pi = PiClass()
```

This means that a name like `pi` can be used in (and imported from) two different files named constants.py, as shown in figure 18.1.

The symbolic constants.py file defines `pi` as an abstract Python object, the sole instance of the `PiClass` class. As the system is developed, various operations can be implemented in this class, which return symbolic rather than numeric results.

There is a natural mapping from this design structure to a directory structure. The top-level directory of the project, called mathproj, contains subdirectories ui and comp; comp in turn contains subdirectories symbolic and numeric; and each of symbolic and numeric contains its own constants.py file.

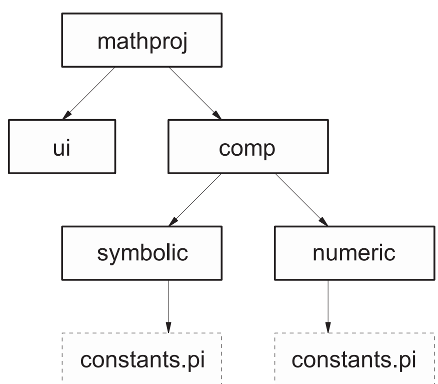


Figure 18.1 Organizing a math package

Given this directory structure, and assuming that the root mathproj directory is installed somewhere in the Python search path, Python code both inside and outside the `mathproj` package can access the two variants of `pi` as `mathproj.symbolic.constants.pi` and `mathproj.numeric.constants.pi`. In other words, the Python name for an item in the package is a reflection of the directory pathname to the file containing that item.

That's what packages are all about. They're ways of organizing very large collections of Python code into coherent wholes, by allowing the code to be split among different files and directories and imposing a module/submodule naming scheme based on the directory structure of the package files. Unfortunately, all isn't this simple in practice because details intrude to make their use more complex than their theory. The practical aspects of packages are the basis for the remainder of this chapter.

18.3 A concrete example

The rest of this chapter will use a running example to illustrate the inner workings of the package mechanism (see figure 18.2). Filenames and paths are shown in plain text, to avoid confusion as to whether we're talking about a file/directory or the module/package defined by that file/directory. The files we'll be using in our example package are shown in listings 18.1 through 18.6.

Listing 18.1 File mathproj/__init__.py

```
print("Hello from mathproj init")
__all__ = ['comp']
version = 1.03
```

Listing 18.2 File mathproj/comp/__init__.py

```
__all__ = ['c1']
print("Hello from mathproj.comp init")
```

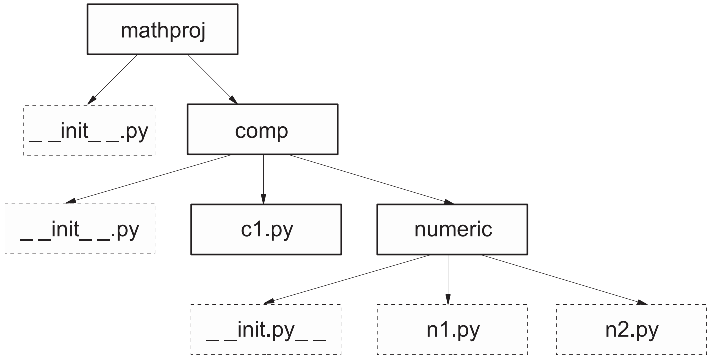


Figure 18.2 Example package

Listing 18.3 File mathproj/comp/c1.py

```
x = 1.00
```

Listing 18.4 File mathproj/comp/numeric/__init__.py

```
print("Hello from numeric init")
```

Listing 18.5 File mathproj/comp/numeric/n1.py

```
from mathproj import version
from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
def g():
    print("version is", version)
    print(h())
```

Listing 18.6 File mathproj/comp/numeric/n2.py

```
def h():
    return "Called function h in module n2"
```

For the purposes of the examples in this chapter, we'll assume that you've created these files in a mathproj directory that's on the Python search path. (It's sufficient to ensure that the current working directory for Python is the directory containing mathproj when executing these examples.)

NOTE In most of the examples in this book, it's not necessary to start up a new Python shell for each example. You can usually execute them in a Python shell you've used for previous examples and still get the results shown. *This isn't true for the examples in this chapter*, because the Python namespace must be clean (unmodified by previous `import` statements) for the examples to work properly. If you do run the examples that follow, please ensure that you run each separate example in its own shell. In IDLE, this requires exiting and restarting the program, not just closing and reopening its Shell window.

18.3.1 Basic use of the mathproj package

Before getting into the details of packages, let's look at accessing items contained in the `mathproj` package. Start up a new Python shell, and do the following:

```
>>> import mathproj
Hello from mathproj init
```

If all goes well, you should get another input prompt and no error messages. As well, the message `"Hello from mathproj init"` should be printed to the screen, by code in the mathproj/__init__.py file. We'll talk more about __init__.py files in a bit; for now, all you need to know is that they're run automatically whenever a package is first loaded.

The mathproj/__init__.py file assigns 1.03 to the variable `version`. `version` is in the scope of the `mathproj` package namespace, and after it's created, you can see it via `mathproj`, even from outside the mathproj/__init__.py file:

```
>>> mathproj.version
1.03
```

In use, packages can look a lot like modules; they can provide access to objects defined within them via attributes. This isn't surprising, because packages are a generalization of modules.

18.3.2 Loading subpackages and submodules

Now, let's start looking at how the various files defined in the `mathproj` package interact with one another. We'll do this by invoking the function `g` defined in the file mathproj/comp/numeric/n1.py. The first obvious question is, has this module been loaded? We have already loaded `mathproj`, but what about its subpackage? Let's see if it's known to Python:

```
>>> mathproj.comp.numeric.n1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'comp'
```

In other words, loading the top-level module of a package isn't enough to load all the submodules. This is in keeping with Python's philosophy that it shouldn't do things behind your back. Clarity is more important than conciseness.

This is simple enough to overcome. We import the module of interest and then execute the function `g` in that module:

```
>>> import mathproj.comp.numeric.n1
Hello from mathproj.comp init
Hello from numeric init
>>> mathproj.comp.numeric.n1.g()
version is 1.03
Called function h in module n2
```

Notice, however, that the lines beginning with `Hello` are printed out as a side effect of loading `mathproj.comp.numeric.n1`. These two lines are printed out by `print` statements in the __init__.py files in mathproj/comp and mathproj/comp/numeric. In other words, before Python can import `mathproj.comp.numeric.n1`, it first has to import `mathproj.comp` and then `mathproj.comp.numeric`. Whenever a package is first imported, its associated __init__.py file is executed, resulting in the `Hello` lines. To confirm that both `mathproj.comp` and `mathproj.comp.numeric` are imported as part of the process of importing `mathproj.comp.numeric.n1`, we can check to see that `mathproj.comp` and `mathproj.comp.numeric` are now known to the Python session:

```
>>> mathproj.comp
<module 'mathproj.comp' from 'mathproj/comp/__init__.py'>
>>> mathproj.comp.numeric
<module 'mathproj.comp.numeric'  from 'mathproj/comp/numeric/__init__.py'>
```

18.3.3 import statements within packages

Files within a package don't automatically have access to objects defined in other files in the same package. As in outside modules, you must use `import` statements to explicitly access objects from other package files. To see how this works in practice, look back at the `n1` subpackage. The code contained in n1.py is

```
from mathproj import version
from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
def g():
    print "version is", version
    print h()
```

`g` makes use of both `versions` from the top-level `mathproj` package and the function `h` from the `n2` module; hence, the module containing `g` must import both `version` and `h` to make them accessible. We import `version` as we would in an `import` statement from outside the `mathproj` package, by saying `from mathproj import version`. In this example, we explicitly import `h` into the code by saying `from mathproj.comp.numeric.n2 import h`, and this will work in any file—explicit imports of package files are always allowed. But because n2.py is in the same directory as n1.py, we can also use a relative import by prepending a single dot to the submodule name. In other words, we can say

```
from .n2 import h
```

as the third line in n1.py, and it works fine.

You can add more dots to move up more levels in the package hierarchy, and you can add module names. Instead of

```
from mathproj import version
from mathproj.comp import c1
from mathproj.comp.numeric.n2 import h
```

we could also have written the imports of n1.py as

```
from ... import version
from ..  import c1
from . n2 import h
```

Relative imports can be handy and quick to type, but be aware that they're *relative* to the module's `__name__`property. Therefore any module being executed as the main module, and thus having an `__name__` of `__main__`, can't use relative imports.

18.3.4 __init__.py files in packages

You'll have noticed that all the directories in our package—mathproj, mathproj/comp, and mathproj/numeric—contain a file called __init__.py. An __init__.py file serves two purposes:

- It's automatically executed by Python the first time a package or subpackage is loaded. This permits whatever package initialization you may desire. Python

requires that a directory contain an __init__.py file before it can be recognized as a package. This prevents directories containing miscellaneous Python code from being accidentally imported as if they defined a package.

- The second point is probably the more important. For many packages, you won't need to put anything in the package's __init__.py file—just make sure an empty __init__.py file is present.

18.4 The __all__ attribute

If you look back at the various __init__.py files defined in mathproj, you'll notice that some of them define an attribute called `__all__`. This has to do with execution of statements of the form `from ... import *`, and it requires explanation.

Generally speaking, we would hope that if outside code executed the statement `from mathproj import *`, it would import all nonprivate names from mathproj. In practice, life is more difficult. The primary problem is that some operating systems have an ambiguous definition of case when it comes to filenames. Microsoft Windows 95/98 is particularly bad in this regard, but it isn't the only villain. Because objects in packages can be defined by files or directories, this leads to ambiguity as to exactly under what name a subpackage might be imported. If we say `from mathproj import *`, will `comp` be imported as `comp`, `Comp`, or `COMP`? If we were to rely only on the name as reported by the operating system, the results might be unpredictable.

There's no good solution to this. It's an inherent problem caused by poor OS design. As the best possible fix, the `__all__` attribute was introduced. If present in an __init__.py file, `__all__` should give a list of strings, defining those names that are to be imported when a `from ... import *` is executed on that particular package. If `__all__` isn't present, then `from ... import *` on the given package does nothing. Because case in a text file is always meaningful, the names under which objects are imported isn't ambiguous, and if the OS thinks that `comp` is the same as `COMP`, that's its problem.

To see this in action, fire up Python again, and try the following:

```
>>> from mathproj import *
Hello from mathproj init
Hello from mathproj.comp init
```

The `__all__` attribute in mathproj/__init__.py contains a single entry, `comp`, and the `import` statement imports only `comp`. It's easy enough to check that `comp` is now known to the Python session:

```
>>> comp
<module 'mathproj.comp' from 'mathproj/comp/__init__.py'>
```

But note that there's no recursive importing of names with a `from ... import *` statement. The `__all__` attribute for the `comp` package contains `c1`, but `c1` isn't magically loaded by our `from mathproj import *` statement:

```
>>> c1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c1' is not defined
```

To insert names from `mathproj.comp` we must, again, do an explicit import:

```
>>> from mathproj.comp import c1
>>> c1
<module 'mathproj.comp.c1' from 'mathproj/comp/c1.py'>
```

18.5 Proper use of packages

Most of your packages shouldn't be as structurally complex as these examples imply. The package mechanism allows wide latitude in the complexity and nesting of your package design. It's obvious that very complex packages *can* be built, but it isn't obvious that they *should* be built.

Here are a couple of suggestions that are appropriate in most circumstances:

- Packages shouldn't use deeply nested directory structures. Except for absolutely huge collections of code, there should be no need to do so. For most packages, a single top-level directory is all that's needed. A two-level hierarchy should be able to effectively handle all but a few of the rest. As written in the *Zen of Python*, by Tim Peters (see the appendix), "Flat is better than nested."
- Although you can use the `__all__` attribute to hide names from `from ... import *` by not listing those names, this is probably *not* a good idea, because it's inconsistent. If you want to hide names, make them private by prefacing them with an underscore.

18.6 Summary

Packages let you create libraries of code that span multiple files and directories, which allows better organization of large collections of code than single modules would allow. Although using packages is useful, you should be wary of nesting directories in your packages more than one or two levels deep, unless you have a very large and complex library.

19 Data types as objects

This chapter covers

- Treating types as objects
- Using types
- Creating user-defined classes
- Understanding duck typing

By now, you've learned the basic Python types and also how to create your own data types using classes. For many languages, that would be pretty much it, as far as data types are concerned. But Python is dynamically typed, meaning that types of things are determined at runtime, not at compile time. This is one of the reasons Python is so easy to use. It also makes it possible, and sometimes necessary, to compute with the types of objects (and not just the objects themselves).

19.1 Types are objects, too

Fire up a Python session, and try out the following:

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

This is the first time you've seen the built-in type function in Python. It can be applied to any Python object and returns the type of that object. In this example, it tells us that 5 is an int (integer) and that ['hello', 'goodbye'] is a list, something you probably already knew.

Of greater interest is that Python returns objects in response to the calls to type; <class 'int'> and <class 'list'> are the screen representations of the returned objects. What sort of object is returned by a call of type(5)? We have any easy way of finding out—just use type on that result:

```
>>> type_result = type(5)
>>> type(type_result)
<class 'type'>
```

The object returned by type is an object whose type happens to be <class 'type'>—we can call it a *type object*. A type object is another kind of Python object whose only outstanding feature is the confusion its name sometime causes. Saying a type object is of type <class 'type'> has about the same degree of clarity as the old Abbot and Costello "Who's on First?" comedy routine.

19.2 Using types

Now that you know that data types can be represented as Python type objects, what can you do with them? You can compare them, because any two Python objects can be compared:

```
>>> type("Hello") == type("Goodbye")
True
>>> type("Hello") == type(5)
False
```

The types of "Hello" and "Goodbye" are the same (they're both strings), but the types of "Hello" and 5 are different. Among other things, you can use this to provide type checking in your function and method definitions.

19.3 Types and user-defined classes

The most common reason to be interested in the types of objects, particularly instances of user-defined classes, is to find out whether a particular object is an instance of a class. After determining that an object is of a particular type, the code can treat it appropriately. An example makes things much clearer. To start, let's define a couple of empty classes, so as to set up a simple inheritance hierarchy:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
```

Now, create an instance of class `B`:

```
>>> b = B()
```

As expected, applying the `type` function to `b` tells us that `b` is an instance of the class `B` that's defined in our current `__main__` namespace:

```
>>> type(b)
<class '__main__.B'>
```

We can also obtain exactly the same information by accessing the instance's special `__class__` attribute:

```
>>> b.__class__
<class '__main__.B'>
```

We'll be working with that class quite a bit to extract further information, so let's store it somewhere:

```
>>> b_class = b.__class__
```

Now, to emphasize that everything in Python is an object, let's prove that the class we obtained from `b` is the class we defined under the name `B`:

```
>>> b_class == B
True
```

In this example, we didn't need to store the class of `b`—we already had it—but I want to make clear that a class is just another Python object and can be stored or passed around like any Python object.

Given the class of `b`, we can find the name of that class using its `__name__` attribute:

```
>>> b_class.__name__
'B'
```

And we can find out what classes it inherits from by accessing its `__bases__` attribute, which contains a tuple of all of its base classes:

```
>>> b_class.__bases__
(<class '__main__.A'>,)
```

Used together, `__class__`, `__bases__`, and `__name__` allow a full analysis of the class inheritance structure associated with any instance.

But two built-in functions provide a more user-friendly way of obtaining most of the information we usually need: `isinstance` and `issubclass`. The `isinstance` function is what you should use to determine whether, for example, a class passed into a function or method is of the expected type:

```
>>> class C:
...     pass
...
>>> class D:
...     pass
...
```

```
>>> class E(D):
...     pass
...
>>> x = 12
>>> c = C()
>>> d = D()
>>> e = E()
>>> isinstance(x, E)
False
>>> isinstance(c, E)                ❶
False
>>> isinstance(e, E)
True
>>> isinstance(e, D)                ❷
True
>>> isinstance(d, E)                    ❸
False
>>> y = 12
>>> isinstance(y, type(5))              ❹
True
The issubclass function is only for class types.
>>> issubclass(C, D)
False
>>> issubclass(E, D)
True
>>> issubclass(D, D)                    ❺
True
>>> issubclass(e.__class__, D)
True
```

For class instances, check against the class ❶. `e` is an instance of class `D` because `E` inherits from `D` ❷. But `d` isn't an instance of class `E` ❸. For other types, you can use an example ❹. A class is considered a subclass of itself ❺.

19.4 Duck typing

Using `type`, `isinstance`, and `issubclass` makes it fairly easy to make code correctly determine an object's or class's inheritance hierarchy. Although this is easy, Python also has a feature that makes using objects even easier: *duck typing*. Duck typing, as in "if it walks like a duck and quacks like a duck, it probably *is* a duck," refers to Python's way of determining whether an object is the required type for an operation, focusing on an object's interface rather than its type. If an operation needs an iterator, for example, the object used doesn't need to be a subclass of any particular iterator or of any iterator at all. All that matters is that the object used as an iterator is able to yield a series of objects in the expected way.

This is in contrast to a language like Java, where stricter rules of inheritance are enforced. In short, duck typing means that in Python, you don't need to (and probably shouldn't) worry about type-checking function or method arguments and the like. Instead, you should rely on readable and documented code combined with thorough testing to make sure an object "quacks like a duck" as needed.

Duck typing can increase the flexibility of well-written code and, combined with the more advanced OO features discussed in chapter 20, gives you the ability to create classes and objects to cover almost any situation.

19.5 Summary

With what we've covered here, you have all the tools necessary to provide type checking in the situations where it's necessary for your applications. On the other hand, by taking advantage of duck typing, you can write more flexible code that doesn't need to be as concerned with type checking. As you'll see in the next chapter, Python's use of duck typing and its flexibility in defining special method attributes make it possible to construct and combine classes in almost any way imaginable.

20 Advanced object-oriented features

This chapter covers

- Using special method attributes
- Making an object behave like a list
- Subclassing built-in types
- Understanding metaclasses
- Creating abstract base classes

This chapter will focus on some more advanced object-oriented features of Python. Python is distinguished by the ability to modify its objects in ways that can fundamentally change their behavior. For C++ users, this is somewhat similar to operator overloading, but in Python it's more comprehensive and easier to use.

In addition to modifying the behavior of objects, you can also control the behavior of classes and the creation of their instances. Obviously, you'll need to be fairly familiar with OO programming to use this feature.

20.1 What is a special method attribute?

A *special method attribute* is an attribute of a Python class with a special meaning to Python. It's defined as a method but isn't intended to be used directly as such. Special methods aren't usually directly invoked; instead, they're called automatically by Python in response to a demand made on an object of that class.

Perhaps the simplest example of this is the `__str__` special method attribute. If it's defined in a class, then anytime an instance of that class is used where Python requires a user-readable string representation of that instance, the `__str__` method attribute will be invoked, and the value it returns will be used as the required string. To see this, let's define a class representing RGB colors as a triplet of numbers, one each for red, green, and blue intensities. As well as defining the standard `__init__` method to initialize instances of the class, we'll also define a `__str__` method to return strings representing instances, in a reasonably human-friendly format. Our definition looks something like listing 20.1.

Listing 20.1 File color_module.py

```
class Color:
    def __init__(self, red, green, blue):
        self._red = red
        self._green = green
        self._blue = blue
    def __str__(self):
        return "Color: R={0:d}, G={1:d}, B={2:d}".format (self._red,
                                                  self._green, self._blue)
```

If we now put this definition into a file called color_module.py, we can load it and use it in the normal manner:

```
>>> from color_module import Color
>>> c = Color(15, 35, 3)
```

The presence of the `__str__` special method attribute can be seen if we now use `print` to print out `c`:

```
>>> print(c)
Color: R=15, G=35, B=3
```

Even though our `__str__` special method attribute has not been explicitly invoked by any of our code, it has nonetheless been used by Python, which knows that the `__str__` attribute (if present) defines a method to convert objects into user-readable strings. This is the defining characteristic of special method attributes—they allow you to define functionality that hooks into Python in special ways. Special method attributes can be used to define classes whose objects behave in a fashion that's syntactically and semantically equivalent to lists or dictionaries. You could, for example, use this ability to define objects that are used in exactly the same manner as Python lists but that use balanced trees rather than arrays to store data. To a programmer, they would appear to be lists, but with faster inserts, slower iterations, and certain other performance differences that would presumably be advantageous in the problem at hand.

The rest of this chapter covers longer examples using special method attributes. It doesn't discuss all of Python's available special method attributes, but it does expose you to the concept in enough detail that you can then easily make use of the other special attribute methods, all of which are defined in the reference appendix.

20.2 Making an object behave like a list

This sample problem involves a large text file containing records of people; each record consists of a single line containing the person's name, age, and place of residence, with a double semicolon (`::`) between the fields. A few lines from such a file might look like this:

```
.
.
.
John Smith::37::Springfield, Massachusetts
Ellen Nelle::25::Springfield, Connecticut
Dale McGladdery::29::Springfield, Hawaii
.
.
.
```

Suppose we need to collect information as to the distribution of ages of people in the file. There are many ways the lines in this file could be processed. Here's one way:

```
fileobject = open(filename, 'r')
lines = fileobject.readlines()
fileobject.close()
for line in lines:
    . . . do whatever . . .
```

That would work in theory, but it reads the entire file into memory at once. If the file was too large to be held in memory (and these files potentially are that large), the program wouldn't work.

Another way to attack the problem is this:

```
fileobject = open(filename, 'r')
for line in fileobject:
    . . . do whatever . . .
fileobject.close()
```

This would get around the problem of too little memory by reading in only one line at a time. It would work fine, but suppose we wanted to make opening the file even simpler and that we wanted to get only the first two fields (name and age) of the lines in the file? We'd need something that could, at least for the purposes of a `for` loop, treat a text file as a list of lines, but without reading the entire text file in at once.

20.2.1 The __getitem__ special method attribute

A solution is to use the `__getitem__` special method attribute, which you can define in any user-defined class, to enable instances of that class to respond to list access syntax and semantics. If `AClass` is a Python class that defines `__getitem__`, and `obj` is an

instance of that class, then things like `x = obj[n]` and `for x in obj:` are meaningful; `obj` may be used in much the same way as a list.

Here's the resulting code; explanations follow:

```
class LineReader:
    def __init__(self, filename):
        self.fileobject = open(filename, 'r')          # Opens file for reading
    def __getitem__(self, index):
        line = self.fileobject.readline()              # Tries to read line
        if line == "":                                 # If no more data ...
            self.fileobject.close()                    # ...closes fileobject....
            raise IndexError                           # ...and raises IndexError
        else:
            return line.split("::")[:2]                # Otherwise, splits line, returns first two fields

for name, age in LineReader("filename"):
    . . . do whatever . . .
```

At first glance, this may look worse than the previous solution because there's more code and it's difficult to understand. But most of that code is in a class, which can be put into its own module, say the `myutils` module. Then the program becomes

```
import myutils
for name, age in myutils.LineReader("filename"):
    . . . do whatever . . .
```

The `LineReader` class handles all the details of opening the file, reading in lines one at a time, and closing the file. At the cost of somewhat more initial development time, it provides a tool that makes working with one-record-per-line large text files easier and less error prone. Note that there are several powerful ways to read files already in Python, but this example has the advantage that it's fairly easy to understand. When you get the idea, you can apply the same principle in many different situations.

20.2.2 How it works

`LineReader` is a class, and the `__init__` method opens the named file for reading and stores the opened `fileobject` for later access. To understand the use of the `__getitem__` method, you need to know the following three points:

- Any object that defines `__getitem__` as an instance method can return elements as if it were a list: all accesses of the form `object[i]` are transformed by Python into a method invocation of the form `object.__getitem__(i)`, which is then handled as a normal method invocation. It's ultimately executed as `__getitem__(object, i)`, using the version of `__getitem__` defined in the class. The first argument of each call of `__getitem__` is the object from which data is being extracted, and the second argument is the index of that data.
- Because `for` loops access each piece of data in a list, one at a time, a `for arg in sequence:` loop works by calling `__getitem__` over and over again, with sequentially increasing indexes. The `for` loop will first set `arg` to `sequence.__getitem__(0)`, then to `sequence.__getitem__(1)`, and so on.

- A `for` loop catches `IndexError` exceptions and handles them by exiting the loop. This is how `for` loops are terminated when used with normal lists or sequences.

The `LineReader` class is intended for use only with and inside a `for` loop, and the `for` loop will always generate calls with a uniformly increasing index: `__getitem__(self, 0)`, `__getitem__(self, 1)`, `__getitem__(self, 2)`, and so on. The previous code takes advantage of this knowledge and returns lines one after the other, ignoring the `index` argument.

With this knowledge, understanding how a `LineReader` object emulates a sequence in a `for` loop is easy. Each iteration of the loop causes the special Python attribute method `__getitem__` to be invoked on the object; as a result, the object reads in the next line from its stored `fileobject` and examines that line. If the line is non-empty, it's returned. An empty line means the end of the file has been reached, and the object closes the `fileobject` and raises the `IndexError` exception. `IndexError` is caught by the enclosing `for` loop, which then terminates.

Remember that this example is here for illustrative purposes only. Usually, iterating over the lines of a file using the `for line in fileobject:` type of loop is sufficient, but this example does show how easy it is in Python to create objects that behave like lists or other types.

20.2.3 Implementing full list functionality

In the previous example, an object of the `LineReader` class behaves like a list object only to the extent that it correctly responds to sequential accesses of the lines in the file it's reading from. You may wonder how this functionality can be expanded to make `LineReader` (or other) objects behave more like a list.

First, the `__getitem__` method should handle its index argument in some way. Because the whole point of the `LineReader` class is to avoid reading a large file into memory, it wouldn't make sense to have the entire file in memory and return the appropriate line. Probably the smartest thing to do would be to check that each index in a `__getitem__` call is one greater than the index from the previous `__getitem__` call (or is 0, for the first call of `__getitem__` on a `LineReader` instance), and to raise an error if this isn't the case. This would ensure that `LineReader` instances are used only in `for` loops as was intended.

More generally, Python provides a number of special method attributes relating to list behavior. `__setitem__` provides a way of defining what should be done when an object is used in the syntactic context of a list assignment, `obj[n] = val`. Some other special method attributes provide less-obvious list functionality, such as the `__add__` attribute, which enables objects to respond to the `+` operator and hence to perform their version of list concatenation. Several other special methods also need to be defined before a class fully emulates a list, but you can achieve this complete list emulation by defining the appropriate Python special method attributes. The next section gives an example that goes further toward implementing a full-list emulation class.

20.3 Giving an object full list capability

`__getitem__` is one of many Python special function attributes that may be defined in a class, to permit instances of that class to display special behavior. To see how this can be carried further, effectively integrating new abilities into Python in a seamless manner, we'll look at another, more comprehensive example.

When lists are used, it's common that any particular list will contain elements of only one type such as a list of strings or a list of numbers. Some languages, such as C++, have the ability to enforce this. In large programs, this ability to declare a list as containing a certain type of element can help you track down errors. An attempt to add an element of the wrong type to a typed list will result in an error message, potentially identifying a problem at an earlier stage of program development than would otherwise be the case.

Python doesn't have typed lists built in, and most Python coders don't miss them; but if you're concerned about enforcing the homogeneity of a list, special method attributes make it easy to create a class that behaves like a typed list. Here's the beginning of such a class (which makes extensive use of the Python built-in `type` and `isinstance` functions, to check the type of objects):

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)                        ❶
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
                if not isinstance(element, self.type):
                    raise TypeError("Attempted to add an element of "
                                    "incorrect type to a typed list.")
        self.elements = initial_list[:]
```

The `example_element` argument defines the type this list can contain by providing an example of the type of element ❶.

The `TypedList` class, as defined here, gives us the ability to make a call of the form

```
x = TypedList('Hello', ["List", "of", "strings"])
```

The first argument, `'Hello'`, isn't incorporated into the resulting data structure at all. It's used as an example of the type of element the list must contain (strings, in this case). The second argument is an optional list that can be used to give an initial list of values. The `__init__` function for the `TypedList` class checks that any list elements passed in when a `TypedList` instance is created are of the same type as the example value given. If there are any type mismatches, an exception is raised.

This version of the `TypedList` class can't be used as a list, because it doesn't respond to the standard methods for setting or accessing list elements. To fix this, we need to define the `__setitem__` and `__getitem__` special method attributes. The `__setitem__` method will be called automatically by Python anytime a statement of the form `TypedListInstance[i] = value` is executed, and the `__getitem__` method

will be called anytime the expression `TypedListInstance[i]` is evaluated to return the value in the *i*th slot of `TypedListInstance`. Here is the next version of the `TypedList` class. Because we'll be type-checking a lot of new elements, we've abstracted this function out into the new private method `__check`:

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        self.elements = initial_list[:]
    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.elements[i] = element
    def __getitem__(self, i):
        return self.elements[i]
```

Now, instances of the `TypedList` class look more like lists. For example, the following code is valid:

```
>>> x = TypedList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
```

The accesses of elements of `x` in the `print` statement are handled by `__getitem__`, which passes them down to the list instance stored in the `TypedList` object. The assignments to `x[2]` and `x[3]` are handled by `__setitem__`, which checks that the element being assigned into the list is of the appropriate type and then performs the assignment on the list contained in `self.elements`. The last line uses `__getitem__` to unpack the first four items in `x` and then pack them into the variables `a`, `b`, `c`, `d`, and `e`, respectively. The calls to `__getitem__` and `__setitem__` are made automatically by Python.

Completion of the `TypedList` class, so that `TypedList` objects behave in all respects like list objects, requires more code. The special method attributes `__setitem__` and `__getitem__` should be defined so that `TypedList` instances can handle slice notation as well as single item access. `__add__` should be defined so that list addition (concatenation) can be performed, and `__mul__` should be defined so that list multiplication can be performed. `__len__` should be defined so that calls of `len(TypedListInstance)` are evaluated correctly. `__delitem__` should be defined so that the `TypedList` class can handle `del` statements correctly. Also, an `append` method

should be defined so that elements can be appended to `TypedList` instances using the standard list-style append, and similarly for an `insert` method.

20.4 Subclassing from built-in types

The previous example makes for a good exercise in understanding how to implement a list-like class from scratch, but it's also a lot of work. In practice, if you were planning to implement your own list-like structure along the lines demonstrated here, you might instead consider subclassing the list type or the `UserList` type.

20.4.1 Subclassing list

Instead of creating a class for a typed list from scratch, as we did in the previous examples, you can also subclass the list type and override all the methods that need to be aware of the allowed type. One big advantage of this approach is that your class has default versions of all list operations, because it's a list already. The main thing to keep in mind is that every type in Python is a class, and if you need a variation on the behavior of a built-in type, you may want to consider subclassing that type:

```
class TypedListList(list):
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        super().__init__(initial_list)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")

    def __setitem__(self, i, element):
        self.__check(element)
        super().__setitem__(i, element)

>>> x = TypedListList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
>>> x[:]
['', '', 'Hello', 'There', '']
>>> del x[2]
>>> x[:]
['', '', 'There', '']
>>> x.sort()
>>> x[:]
['', '', '', 'There']
```

Note that all that we need to do in this case is implement a method to check the type of items being added and then tweak `__setitem__` to make that check before calling `list`'s regular `__setitem__` method. Other methods, like `sort` and `del`, work without any further coding. Overloading a built-in type can save a fair amount of time if you need only a few variations in its behavior, because the bulk of the class can be used unchanged.

20.4.2 Subclassing UserList

If you need a variation on a list (as in the previous examples), there's a third alternative. You can subclass the `UserList` class, a list wrapper class found in the `collections` module. `UserList` was created for earlier versions of Python when subclassing the list type wasn't possible; but it's still useful, particularly in our current situation, because the underlying list is available as the `data` attribute:

```
from collections import UserList
class TypedUserList(UserList):
    def __init__(self, example_element, initial_list=[]):
        super().__init__(initial_list)
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.data[i] = element
    def __getitem__(self, i):
        return self.data[i]

>>> x = TypedUserList("", 5 * [""])
>>> x[2] = "Hello"
>>> x[3] = "There"
>>> print(x[2] + ' ' + x[3])
Hello There
>>> a, b, c, d, e = x
>>> a, b, c, d
('', '', 'Hello', 'There')
>>> x[:]
['', '', 'Hello', 'There', '']
>>> del x[2]
>>> x[:]
['', '', 'There', '']
>>> x.sort()
>>> x[:]
['', '', '', 'There']
```

This is much the same as subclassing `list`, except that in the implementation of the class, the list of items is available internally as the `data` member. In some situations, having direct access to the underlying data structure can be useful; and in addition to `UserList`, there are also `UserDict` and `UserString` wrapper classes.

20.5 When to use special method attributes

As a rule, it's a good idea to be somewhat cautious with the use of special method attributes. Other programmers who need to work with your code may wonder why one sequence-type object responds correctly to standard indexing notation, whereas another doesn't.

My general guidelines are to use special method attributes in either of two situations. First, if I have a frequently used class in my own code that behaves in some respects like a Python built-in type, I'll define such special method attributes as useful. This occurs most often with objects that behave like sequences in one way or another. Second, if I have a class that behaves identically or almost identically to a built-in class, I may choose to define all of the appropriate special function attributes or subclass the built-in Python type and distribute the class. An example of the latter might be lists implemented as balanced trees so that access is slower but insertion is faster than with standard lists.

These aren't hard-and-fast rules. For example, it's often a good idea to define the `__str__` special method attribute for a class, so that you can say `print(instance)` in debugging code and get an informative and nice-looking representation of your object printed to the screen.

20.6 Metaclasses

Everything in Python is an object, including classes. An object has to be created from something, and in the case of a class it's created from a *metaclass*. In Python, classes are objects that are created at runtime as instances of the metaclass `type`. Let's look at the standard definition of a class:

```
class Spam:
    def __init__(self, x):
        self.x = x
    def show(self):
        print(self.x)
```

This is the ordinary way of creating a class. We can create instances of it and exercise them and so on:

```
>>> my_spam = Spam("test")
>>> type(my_spam)
<class '__main__.Spam'>
>>> type(Spam)
<class 'type'>
>>> my_spam.show()
test
```

Note that the type of the class `spam` is `'type'`.

Although this example is the common way to create a class, it's really a shortcut for creating it explicitly using a metaclass. To do so, you need to call the metaclass (`type`, by default) with the name of the class, a tuple of its base classes, and a dictionary of its attributes:

```
def init(self, x):
    self.x = x
def show(self):
    print(self.x)
Spam = type("Spam", (object,), {'__init__': init, 'show': show})
```

This creates exactly the same class as before, with an `__init__` and a `show` method:

```
>>> my_spam = Spam("test")
>>> type(my_spam)
<class '__main__.Spam'>
>>> type(Spam)
<class 'type'>
>>> my_spam.show()
test
```

It looks a bit strange, because the methods are defined as first-class functions; but the result is the same, and the type of `spam` is still `'type'`.

The point of this exercise is that the `type` metaclass can be subclassed and its behavior can be changed. That means that the way classes themselves are created from objects can be modified, allowing you to create classes that register or verify their instances when they're created, for example, or classes that automatically have a class attribute. In the following somewhat simple-minded example, `type` is subclassed to `NewType`, which announces when it creates a class object and adds a class attribute `new_attr` to that class:

```
class NewType(type):
    def __init__(cls, name, bases, methods):
        print("Creating from NewType")
        type.__init__(cls, name, bases, methods)
        cls.new_attr = "test"

def init(self, x):
    self.x = x

def show(self):
    print(self.x)

Spam = NewType("Spam", (object,), {'__init__': init, 'show': show})
Creating from NewType
>>> Spam.new_attr
'test'
>>> my_spam = Spam("test")
>>> type(Spam)
<class '__main__.NewType'>
>>> type(my_spam)
```

```
<class '__main__.Spam'>
>>> my_spam.show()
test
```

It's not necessary to use the long form of class creation in order to use a custom metaclass, however. You can accomplish the same thing by using the `metaclass` keyword with a simple class definition:

```
class NewType(type):
   def __init__(cls, name, bases, dict):
       print("Creating from NewType")
       cls.new_attr = "test"
       type.__init__(cls, name, bases, dict)

class Spam(metaclass=NewType):
    def __init__(self, x):
        self.x = x
    def show(self):
        print(self.x)

Creating from NewType
>>> Spam.new_attr
'test'
>>> my_spam = Spam("test")
>>> type(Spam)
<class '__main__.NewType'>
>>> type(my_spam)
<class '__main__.Spam'>
>>> my_spam.show()
test
```

The previous examples have been kept deliberately simple, with the aim of making the basic mechanics of using metaclasses clear. Metaclass programming is enormously powerful, but it's also a complex topic, and its details and use cases are well beyond both this book and most coding needs. In the words of master Pythonista Tim Peters, "Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who need them know with certainty that they need them, and don't need an explanation about why)."

20.7 Abstract base classes

As you've seen, Python's strategy for interacting with an object is to invoke its methods and judge its type from what it does. This use of duck typing is also referred to as EAFP, which is short for "easier to ask forgiveness than permission." The common approach in Python is to invoke an object's method and either succeed or deal with the exception. If an object behaves like a sequence, for example, you can iterate over it with a `for` loop. If not, you catch and deal with the `TypeError` exception. Although this approach works well most of the time, there are occasions where it's better for your code to know exactly what it's getting into. This is sometimes called the LBYL, or "look before you leap," approach.

For example, suppose we need to know whether an object is a mutable sequence—something that works like a list. We can use isinstance to see if the object has list as a base class:

```
if isinstance(my_object, list):
    # do stuff
```

Or we can see if it has a __getitem__ method defined, by accessing the object with [], for example:

```
try:
    x = my_object[0]
    #do stuff
except TypeError:
    pass
```

The problem is that the first approach will miss perfectly good mutable sequences, like our LineReader class at the beginning of this chapter, because they aren't subclasses of list. The second approach, on the other hand, will go in the other direction and accept both tuples (which aren't mutable) and dictionaries (which aren't sequences), among others.

Although it's not in harmony with the overall philosophy of Python, in some scenarios it's useful to be able to know for sure that an object is a sequence, a mutable sequence, a mapping, and so on. Python's answer is an abstract base class (ABC), which is a class that can be put into an object's inheritance tree to indicate to an external inspector that the object has a certain set of features. You can then test objects using isinstance for the presence of that abstract base class. The collections library contains several abstract collection types, including the following:

```
'Hashable', 'Iterable', 'Mapping', 'MutableMapping', 'MutableSequence',
'MutableSet', 'Sequence', 'Sized'
```

There are also abstract base classes for various other sorts of objects—mapping, sets, and so on.

20.7.1 Using abstract base classes for type checking

To return to our example, let's see how we can use an abstract base class to make sure the custom TypedList class we created previously is identified as a mutable sequence. The first things we need to do are to import the MutableSequence base class from the collections module and then register our class as a MutableSequence:

```
class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        self.elements = initial_list[:]
```

```
    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.elements[i] = element
    def __getitem__(self, i):
        return self.elements[i]
>>> from collections import MutableSequence
>>> issubclass(TypedList, MutableSequence)
False
>>> MutableSequence.register(TypedList)
>>> issubclass(TypedList, MutableSequence)
True
>>> my_list = TypedList(1)
>>> isinstance(my_list, MutableSequence)
True
```

When `TypedList` is registered with `MutableSequence` as one of its own, any instance of it will also be an instance of `MutableSequence`.

20.7.2 Creating abstract base classes

You can also create your own abstract base classes by setting their metaclass to be `ABCMeta` from the `abc` module. For example, if we want to make sure that every instance of `list` was also identified as an instance of `MyABC`, we do the following:

```
>>> from abc import ABCMeta
>>> class MyABC(metaclass=ABCMeta):
...     pass
...
>>> MyABC.register(list)
>>> isinstance([1, 2, 3], MyABC)
True
```

Being able to use abstract base classes in Python gives you a choice: you can look before you leap, or you can ask for forgiveness.

20.7.3 Using the @abstractmethod and @abstractproperty decorators

In Java, for example, an abstract class by definition can't be instantiated under any circumstances. As with many features, in Python the "abstractness" of abstract base classes isn't so much an enforced rule as it is a gentleman's agreement. Python will allow you to create instances of generic abstract base classes without complaint as long as the base class doesn't contain an abstract method:

```
>>> from abc import ABCMeta
>>> class MyABC(metaclass=ABCMeta):
...     pass
...
>>> my_myabc = MyABC()
>>> print(type(my_myabc))
<class '__main__.MyABC'>
```

But if the class has an abstract method, then that class can't be instantiated, nor can any subclass be instantiated unless it has overridden the abstract method. To create an abstract method, you use the `@abstractmethod` decorator from the `abc` module:

```
>>> from abc import ABCMeta, abstractmethod
>>> class MyABC(metaclass=ABCMeta):
...     @abstractmethod
...     def overrideme(self):
...         print("in ABC")
...
>>> my_myabc = MyABC()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MyABC with abstract
➥ methods overrideme
>>> class SecondABC(MyABC):
...     pass
>>> my_secondabc = SecondABC()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class SecondABC with abstract
➥ methods overrideme
```

As the exceptions indicate, it's not possible to instantiate a class with an abstract method, nor is it possible to instantiate a subclass, unless the abstract method has been overridden. In the following example, the abstract method has been overridden for the class `SecondABC` so the class can now be instantiated:

```
>>> class SecondABC(MyABC):
...     def overrideme(self):
...         super().overrideme()
...         print("in SecondABC")
...
>>> my_secondabc = SecondABC()
>>> my_secondabc.overrideme()
in ABC
in SecondABC
```

The `abstractmethod` function sets a function attribute `__isabstractmethod__`, which is checked by the `__new__` method of `ABCMeta`. This means you can create abstract methods only for classes that are created with `ABCMeta` as their metaclass, and they must be defined in the class definition, not dynamically added.

Abstract methods in Python, unlike in Java, can have an implementation. You can call those methods by the overriding method in a subclass, as happens in the previous example.

There is also an `@abstractproperty` decorator, which adds abstract properties to a class. These work the same way as normal properties, except that a class containing an abstract property can't be instantiated, nor can its subclasses, unless they override the abstract property with a concrete one:

```
from abc import ABCMeta, abstractproperty
class MyABC(metaclass=ABCMeta):
```

```
    @abstractproperty
    def readx(self):
        return self.__x
    def getx(self):  # read only
        return self.__x
    def setx(self, x):
        self.__x = x
    x = abstractproperty(getx, setx)
```

Even though the property `x` created here has an implementation that would work, this class can't be instantiated. Instead, it must be subclassed, and the abstract property `x` must be overridden. But the subclass can access the abstract property in `MyABC`.

20.8 Summary

Python has several object-oriented options. By adding special method attributes to your classes, you can simulate other classes, even built-in types. Or you can subclass any of Python's built-in types to modify their behavior as needed. You can even control the creation and behavior of classes themselves by creating or using different metaclasses. If those options aren't enough, you can also use abstract base classes to customize the possibilities for type checking.

Part 4

Where can you go from here?

Part 4 introduces features of Python beyond bare syntax and syntax and coding. Starting with the basics of testing in Python, we'll also look at migration to Python 3 from Python 2.x and tour the standard library. The final chapter uses several examples and a simple project to illustrate the power of Python for creating database-driven web applications.

21 Testing your code made easy(-er)

This chapter covers

- Testing your code
- Debugging with the `assert` statement
- Using Python's debug variable
- Testing using docstrings
- Creating and using unit tests

The problem with writing code is that you're never sure you've got it right. Every time you turn around, bugs crop up, and what's worse, fixing those bugs is likely to create more bugs. Fortunately, Python encourages readable code, which helps in debugging; but we still need all the help we can get in maintaining our code.

21.1 Why you need to have tests

Almost all code needs maintenance. Sometimes it requires minor bug fixes, but other times it needs major changes or additions, or even a complete redesign. The more you change your code, the more likely you are to inadvertently introduce new

problems or mistakes. What you need is a way to make sure you don't create new bugs when you fix the old ones and that everything still works when you redesign and refactor and improve your code. You need ways to verify that what used to work still works. You need tests.

21.2 The assert statement

The quickest way to put a test into your code is with the assert statement. An assert statement is a simple way of putting a conditional in your code that will raise an exception if an expression isn't true. That makes it an ideal watchdog for situations where a particular precondition must always be true for the code to function correctly. For example, see the file in listing 21.1.

Listing 21.1 File assert_test.py

```
def example(param):
    """param must be greater than 0!"""
    assert param > 0          # <-- Checks value of param
    # do stuff here

>>> import assert_test
>>> assert_test.example(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "assert_test.py", line 2, in example
    assert param > 0
AssertionError
>>>
```

21.2.1 Python's __debug__ variable

Although assert statements are a bit more streamlined, by themselves they're conditionals that raise exceptions when their expressions are false. It's a fair concern that using assert statements generously will leave code littered with extra conditional statements that will impact its performance. The assert statement relies on a built-in variable in Python, __debug__, which is True by default. That means the assert statement we used previously in assert_test.py is equivalent to

```
if __debug__:
    if not param > 0:
        raise AssertionError
```

But if the __debug__ variable is False, no code will be generated at all for assert statements. The catch is that __debug__ can't be directly assigned:

```
>>> __debug__
True
>>> __debug__ = False
  File "<stdin>", line 1
SyntaxError: assignment to keyword
```

To turn off the __debug__ variable, either you need to have the PYTHONOPTIMIZE environment variable set, or you need to run Python with the -O option. When the

`__debug__` variable has been turned off with the `-O` parameter, the previous test using assert_test.py no longer gives an error:

```
>>> import assert_test
>>> assert_test.example(0)
>>>
```

By using `assert` (and the `__debug__` variable), you can have several checks in place as you develop and test your code. And then, by giving Python a single option, you can have all of that testing code disappear for production runs but still be available the next time you need to debug.

21.3 Tests in docstrings: doctests

Using `assert` statements is simple and relatively easy but also rather limited. Although `assert` statements can check specific spots in your code, they give no support for creating more complete suites of tests that can be run to test entire modules. Python has an easy way to test your code that uses the docstrings you should already be including in your interactive sessions, testing some code that's cut and pasted into the docstring for that code.

For example, let's return to the `TypedList` class we created in the last chapter. As you may recall, the idea was to create a list-like class that allowed only a single type of item. Because we had to add `__getitem__` and `__setitem__` special methods, it would be good to test them to make sure that they work as expected when we use `[]` to access items. Using a Python shell, we can do something like this:

```
>>> from typedlist import TypedList
>>> a_typed_list = TypedList(1, [1, 2, 3])
>>> a_typed_list[1] == 2
True
>>> a_typed_list[1] = 3
>>> a_typed_list[1]
3
>>>
```

This interactive session verifies that with an index of 1, both `__getitem__` and `__setitem__` work correctly to access the second item of a `TypedList`. To make this session into a doctest, we can copy and paste it into the docstring of the `typedlist` module and add code to run the test if the module is executed directly as a script. See listing 21.2.

Listing 21.2 File typedlist_doctest.py

```
""" a list that only allows items of a single type

any text (like this) that isn't in shell format is ignored by doctest

>>> from typedlist_doctest import TypedList
>>> a_typed_list = TypedList(1, [1, 2, 3])
>>> a_typed_list[1] == 2
True
```

```
>>> a_typed_list[1] = 3
>>> a_typed_list[1]
3
>>>
"""

class TypedList:
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        self.elements = initial_list[:]
    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
                            "incorrect type to a typed list.")
    def __setitem__(self, i, element):
        self.__check(element)
        self.elements[i] = element
    def __getitem__(self, i):
        return self.elements[i]

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

If we run `typedlist_doctest.py` from a command prompt, by default it prints nothing if all the tests pass. But if a test fails, it's reported in detail. For example, let's change the first access of `a_typed_list[1]` to expect a 3 instead of a 2:

```
>>> a_typed_list[1] == 3
True
```

Now the test will fail, and doctest will report that clearly:

```
**********************************************************************
File "typedlist_doctest.py", line 7, in __main__
Failed example:
    a_typed_list[1] == 3
Expected:
    True
Got:
    False
**********************************************************************
1 items had failures:
   1 of   5 in __main__
***Test Failed*** 1 failures.
```

If you need a full report of all tests, both failing and passing, you can make doctest give a verbose report by adding the -v switch after the filename on the command line.

21.3.1 Avoiding doctest traps

Because doctests expect a character-by-character match for a successful test, you'll sometimes find tests failing unexpectedly. In particular, dictionaries aren't guaranteed to print in a particular order. If you had a test like this

```
>>> my_dict = {'one': 1, 'two': 2}
>>> my_dict
{'one': 1, 'two': 2}
```

the two items could conceivably print in either order, causing the test to fail unpredictably. In cases like this, a direct comparison is more reliable:

```
>>> my_dict == {'one': 1, 'two': 2}
True
>>> my_dict == {'two': 2, 'one': 1}
True
```

Similarly, printing object addresses will also cause failures, because it's unlikely that an object will have the same address for two different runs of the test.

It's also important to note that if you want blank lines to be considered part of the output, you need to indicate them with a line containing just `<BLANKLINE>`, because a blank line is normally a signal to doctest of the end of output.

Finally, if you use the \ character, either to escape a character or to continue a line in a doctest, you need to make sure that the docstring is a raw string. Prepending an `r` to the docstring will prevent the \ from being interpreted as part of the string.

21.3.2 Tweaking doctests with directives

Several directives can also tweak the way lines are handled. The most commonly used of these directives are `NORMALIZE_WHITESPACE` and `ELLIPSIS`. The former treats all sequences of whitespace as equal, so that differently spaced sequences of items, or even sequences with line breaks, still pass the test. Similarly, `ELLIPSIS` signals that a sequence of ... will match any substring in the output. Using `ELLIPSIS` can alleviate problems like those mentioned earlier in printing object addresses, if you need to include data that changes from run to run in your doctest. You employ directives by adding them to a `# doctest:` comment following the test, with a + to activate and a - to deactivate them. Directives apply only to a single example, and you can combine multiple directives, either on the same line or on multiple lines:

```
>>> print([1, 2, 3, 4]) # doctest: +ELLIPSIS, +NORMALIZE_WHITESPACE
```

This line matches all of the following:

```
[1,     2,    3,             4]
[1,   2,
  3, 4]
[1, ... 4]
```

Additional directives control the style of output, the treatment of exception tracebacks, and so on, and it's also possible to add your own directives and subclass doctest internals to change the response to them.

21.3.3 Pros and cons of doctests

Doctests have two big advantages going for them: simplicity and ease of use. These virtues make doctests a good thing to use in your code, because simple, easy tests are more likely to be created, used, and maintained. In addition, having the test readily visible and accessible as part of the code is a help. If you can see it and edit it easily, it's more likely that you'll keep it current and use it often. I might add that doctests are a "Pythonic" method of testing.

On the other hand, doctests aren't for everyone or for every situation. If you need to do many tests, docstrings will add considerable bulk to your code files unless you move them to separate files, in which case you've lost one of the advantages of using doctests. In addition, some aspects of your code may be more awkward to test in the interpreter. There's some debate about how far you can take doctests as a testing mechanism. The Zope 3 project, for example, uses doctests extensively, whereas most projects of similar size use unit tests. It finally comes down to what works best for you and your project.

21.4 Using unit tests to test everything, every time

In addition to doctests, Python also has a full-blown unit test library as part of the standard library. The `unittest` module (originally named `PyUnit`) is modeled somewhat on the JUnit library widely used in Java. You can use `unittest` to create a comprehensive suite of tests for any project. Python itself uses unit tests to test its more than 110 modules, including `unittest`. It's not the intent of this chapter to tell you why you should use `unittest` or exactly what you should test but to give you a quick example of how to create and run a `unittest` suite.

The two basic classes you use to create unit tests are `TestCase` and `TestSuite`. The former contains the individual tests, and the latter is used to aggregate tests that should be run together.

21.4.1 Setting up and running a single test case

You create a test, or group of tests, by subclassing `TestCase` and adding each test as a method. To make the tests, you can either use `assert` or use one of the many variations on `assert` that are part of the `TestCase` class. In addition to adding the tests themselves as methods to the subclass, you can override both the `setUp` and `tearDown` methods to handle creating and disposing any objects or conditions needed for the test.

It will be easier to see how test creation works by following a simple example. In the last chapter, we created a `TypedList` class that ensures that all of its items are of the same type. Let's create a simple test case to make sure the `__getitem__` method returns the correct value. See listing 21.3.

Listing 21.3 File testtypedlist.py

```
import unittest
from typedlist import TypedList

class TestTypedList(unittest.TestCase):
    def setUp(self):
        self.a_typedlist = TypedList(1, [1, 2, 3])

    def testGetItem(self):
        self.assertEqual(self.a_typedlist[1], 2)

    def testSetItem(self):
        self.a_typedlist[1] = 3
        self.assertEqual(self.a_typedlist[1], 3)

if __name__ == '__main__':
    unittest.main()
```

In these tests, we use the `assertEqual` method of `TestCase`. As the name implies, `assertEqual` tests for the equality of two values. There are number of test `asserts` in `TestCase`, but you don't absolutely have to use them. You can, for example, also use the regular `assert` statement, if you want. The problem is that if the `__debug__` variable is set to `false`, the assertion won't be tested, and your testing won't occur. Therefore, it's probably wisest always to use the methods in `TestCase`. Table 21.1 lists the main forms of those methods.

Table 21.1 Test methods in `unittest.TestCase`

Test	Explanation
`TestCase.assert(expr[, msg])`	Fails if `expr` is `False`
`TestCase.assertEqual(first, second[, msg])`	Fails if `first` isn't equal to `second`
`TestCase.assertNotEqual(first, second[, msg])`	Fails if `first` is equal to `second`
`TestCase.assertAlmostEqual(first, second[, places[, msg]])`	Fails if `first` and `second` aren't equal when rounded to `places` (default is 7) decimal places
`TestCase.assertNotAlmostEqual(first, second[, places[, msg]])`	Fails if `first` and `second` are equal when rounded to `places` (default is 7) decimal places
`TestCase.assertRaises(exception, callable, ...)`	Passes if calling `callable` with any parameters passed with it raises exception
`TestCase.assertFalse(expr[, msg])`	Fails if `expr` is `True`
`TestCase.fail([msg])`	Unconditionally generates a failure

This is a summary of the most common methods. There are variations with the opposite names: `failUnless`, `failUnlessEqual`, and so on; see the standard documentation for a complete list.

21.4.2 Running the test

To run the test from the command line, we can add a call to the `main` method of the test case:

```
if __name__ == '__main__':
    unittest.main()
```

The output from running this is something like the following:

```
doc@mac:~/work/quickpythonbook/testcode$ python3.1 testtypedlist.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

On the other hand, if a test fails, we get a fuller report. Let's assume that we changed the value of the `__getitem__` test so that it would fail:

```
testGetItem (testtypedlist.TestTypedList) ... FAIL
testSetItem (testtypedlist.TestTypedList) ... ok

======================================================================
FAIL: testGetItem (testtypedlist.TestTypedList)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/doc/work/quickpythonbook/testcode/testtypedlist.py", line 9,
➥ in testGetItem
    self.assertEqual(self.aTypedList[1], 3)
AssertionError: 2 != 3

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Each failure is reported clearly, and the total number of failures is also reported.

21.4.3 Running multiple tests

It's also fairly easy to aggregate various test cases into a unified test suite that can be run with a single command. This is done most easily by using the `TestSuite`, `TestLoader`, and `TestRunner` classes. Although the `TestSuite` class can be subclassed and customized, and test cases can be added to a test suite instance manually, for most applications it's easier to use the module's default instances of `TestLoader` and `TextTestRunner` to create and run a test suite as follows:

```
suite = unittest.defaultTestLoader().loadTestsFromTestCase(TestTypedList)
unittest.TextTestRunner(verbosity=2).run(suite)
```

The previous code uses the module's instance of `defaultTestLoader` to add all tests of the type `TestTypedList` to the test suite called `suite`. Then, the module's default `TextTestRunner` instance runs the suite of tests with a verbosity level of 2. The result looks something like this:

```
testGetItem (__main__.TestTypedList) ... ok
testSetItem (__main__.TestTypedList) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

There are several variations on how test cases can be detected and loaded. For example, to load all tests from the module `testtypedlist.py`, you can use `loadTestsFromName('testtypedlist')`, which loads all tests in classes derived from `TestCase` in that module.

This example has been simple because my intent isn't to tell you how to use unit tests but to show you the basics of how to create a unit test in Python.

21.4.4 Unit tests vs. doctests

Unit tests have a different approach than doctests. Although doctests are by nature intended (if not required) to be interleaved with the code they test, unit tests are meant to be separate from the tested code. This makes unit tests a bit less transparent and convenient in smaller projects, but they also have benefits. Having the tests separate from the code means you can develop an extensive suite of tests without increasing the size of the code files and burying the working code under a mass of test code. It also means that the tests don't need to be distributed with the code, although in many projects they are. On the other hand, the separation of tests and code and the more programmatic nature of unit tests tend to make for less ease of documentation than is found in doctests.

21.5 Summary

Python comes with two different testing systems, doctests and unit tests. Doctests are meant to be included in docstrings for modules, functions, and so on and are more visible and easier to edit. Particularly for small projects, doctests are more visible and easier to use and therefore more likely to be used. Unit tests, on the other hand, are intended for creating more intensive test suites and are more customizable; the tests are separate from the code, arguably making them a better choice for larger projects.

22 Moving from Python 2 to Python 3

This chapter covers

- Using Python 2.6 for preliminary testing
- Converting using `2to3`
- Testing and common pitfalls
- Using the same code base for Python 2 and 3

We've dealt with only the syntax and features of Python 3.x so far in this book. That was deliberate, because we feel that Python 3.x is a distinct improvement over early versions of Python and it's where future development should occur. On the other hand, it's not always possible to make a clean break from the past. For the foreseeable future, many situations will call for dealing with legacy Python 2.x code. This chapter discusses how you can migrate older Python 2.x code to Python 3.x.

22.1 Porting from 2 to 3

Several changes from Python 2.x to 3.x broke compatibility between the two versions. Some of them were obvious but easy to fix, such as the change of the `print` statement to the `print` function or the change of the `input` function to behave the

way that the old raw_input function did. Other changes were more subtle, and more posed sneakier problems: the change of strings to be Unicode by default with the addition of a bytes type to hold unencoded byte values, returning iterators and views instead of lists from functions like range and a dictionary's keys method, differences in handling exceptions, and the addition of sets come to mind, to name a few.

Table 22.1 illustrates some of the representative differences between Python 2.x and 3.x code.

Table 22.1 Python 2.x vs. Python 3.x code

Python 2.x	Python 3.x
`raw_input(prompt)`	`input(prompt)`
`print x` (no parentheses)	`print(x)`
`str` type and `unicode` type	`bytes` type (sequence of bytes) and `str` type (all strings are Unicode)
`1L` (long integer data type)	`1` (all ints can be long)
`an_iterator.next()`	`next(an_iterator)`
`try.... except Exception, e`	`try.... except Exception as e`
`StandardError` class	`Exception` class
`1/2 -> 0` (integer division)	`1/2 -> 0.5, 1//2 -> 0`
`my_dict.keys()` (and many others) return lists	`my_dict.keys()` (and others) return dynamic view that changes if underlying object changes

Although these differences aren't in themselves that large, taken together they mean that virtually no legacy Python 2.x code can run on Python 3.x unchanged.

22.1.1 Steps in porting from Python 2.x to 3.x

To help you make the transition from 2 to 3, the core developers of Python offer a clear set of steps to follow to migrate code to the new version:

1. *Make sure the existing code has adequate test coverage.*
   The change from Python 2.x to Python 3 isn't huge, but there are enough subtle incompatibilities that code will need to be tested to make sure it still behaves as it's supposed to. Complete test coverage isn't recommended—it's necessary.
2. *Test the code using Python 2.6 and the* -3 *command-line switch to find and remove obvious compatibility problems.*
   Running code under Python 2.6 (or later) using the -3 switch will point out some obvious issues that should be fixed before going on.

3 *Run code through the* `2to3` *conversion tool to create a new version of the code with several fixes automatically applied.*

   The `2to3` tool will automatically fix the straightforward incompatibilities, like `print` and `raw_input`, and it will flag some others that it can't fix.

4 *Run the tests on the new code using Python 3.x, and fix the failures.*

   Running the code's tests is likely to yield failures. The tests may well fail to run at all without fixing both the code and the tests. Fix and test until all tests pass.

In general, the first step, complete test coverage, is the biggest stumbling block for most projects. But the time you spend adding tests will continue to pay off long after the conversion is complete.

22.2 Testing with Python 2.6 and -3

Python 2.6 and later versions have a `-3` command-line switch that will report obvious incompatibilities in your code. In listing 22.1, we consider a small Python 2.x script, with several elements that need to be changed to run correctly on Python 3.

Listing 22.1 File convert2_3.py

```
""" example of conversion from Python 2.x to Python 3

"""

def write_file(filename, version, data):
    try:
        outfile = open(filename, "wb")
        outfile.write("%s\n" % (version))
        outfile.write(data)
        outfile.close()

    except StandardError, e:
        print e

def read_file(filename):
    infile = open(filename, "rb")
    file_version = infile.readline()
    data = infile.read()
    major_version = int(file_version[0])
    minor_version = int(file_version[2])

    if major_version <> 1 or minor_version > 5:
        raise Exception, "Wrong file version"

    infile.close()
    return file_version, data

if __name__ == "__main__":
    version = "1.1"
    filename = raw_input("Please enter a filename: ")
    write_file(filename, version, "this is test data")
    print "File created, reading data from file"
```

```
    new_version, data = read_file(filename)
    cents = 73L
    quarters = cents / 25
    print "%d cents contains %d quarters" % (cents, quarters)

    new_dict = {}
    if not new_dict.has_key(new_version):
        new_dict[new_version] = filename
```

This code doesn't do much of anything, but it does contain a lot of fairly common bits of code: getting a string from the user, opening a file, reading from the file, comparing the contents of the file to a string, appending a string, handling errors, and so on. Running this code using Python 2.6 with the `-3` switch generates a couple of deprecation warnings about using `<>` and `dict.has_key`:

```
doc@mac:~$ python2.6 -3 convert2_3.py
convert2_3.py:22: DeprecationWarning: <> not supported in 3.x; use !=
  if major_version <> 1 or minor_version > 5:
Please enter a filename: testfile
File created, reading data from file
convert2_3.py:35: DeprecationWarning: classic int division

  quarters = cents / 25
73 cents contains 2 quarters
convert2_3.py:39: DeprecationWarning: dict.has_key() not supported in 3.x;
➥ use the in operator
  if not new_dict.has_key(new_version):
```

Obviously, some other fixes need to be made. Both `print` and `raw_input` need to be changed, and the exception handling needs to be tweaked, but those issues can be taken care of automatically. Although some of the warnings generated by the `-3` switch can also be corrected by the conversion tool, it's recommended that you fix all of its warnings before taking the next step. In this case, that means changing these lines:

- In the `read_file` function, the `<>` should be replaced by `!=`.
- `cents = 73L` needs to become `73`, because there is no long an integer type in Python 3.x.
- `cents / 25` needs to be changed to `cents // 25` to return an integer number of quarters.
- `if not new_dict.has_key(new_version):` needs to be rephrased to `if new_version not in new_dict:` to eliminate the use of `has_key`.

After we make those changes, the code should run without warnings.

22.3 Using 2to3 to convert the code

After you've fixed any warnings from running the code with the `-3` parameter under Python 2.6 or higher, the next step is to run the `2to3` tool on it to produce a version for Python 3. `2to3` will make a number of automatic fixes, things like changing `raw_input` to `input`, `print` to `print()`, and so on, but it's unlikely that the converted code will run without error.

The `2to3` tool takes either a file or a directory to convert and generates a patch file of all the changes to be applied to convert to Python 3. If it's unable to apply a fix for a problem automatically, it prints a warning below the diff for that file. By default, `2to3` runs several fixers and more can be added; running `2to3 -l` (that's a lowercase "el") will show what fixers are available. If you want to exclude a particular fixer, using the `-x` option followed by the fixer name will exclude it, as in the following example where the `has_key` fixer is turned off:

```
doc@mac:~$ 2to3 -x has_key convert2_3.py
```

Using the `-f` option turns on fixers explicitly, with `all` enabling all the default fixers, whereas using the fixer name enables only the fixer mentioned. If we want to enable only the `has_key` fixer, we use

```
doc@mac:~$ 2to3 -f has_key convert2_3.py
```

And if we want to enable all the default fixers and the `apply` fixer, the command is

```
doc@mac:~$ 2to3 -f all -f apply convert2_3.py
```

Running `2to3` on our sample file gives us the diff in listing 22.2.

Listing 22.2 File convert2_3.diff

```
--- convert2_3.py (original)
+++ convert2_3.py (refactored)
@@ -9,8 +9,8 @@
         outfile.write(data)
         outfile.close()

-    except StandardError, e:
-        print e
+    except Exception as e:
+        print(e)

 def read_file(filename):
     infile = file(filename, "rb")
@@ -19,24 +19,24 @@
     major_version = int(file_version[0])
     minor_version = int(file_version[2])

-    if major_version <> 1 or minor_version > 5:
-        raise Exception, "Wrong file version"
+    if major_version != 1 or minor_version > 5:
+        raise Exception("Wrong file version")

     infile.close()
     return file_version, data

 if __name__ == "__main__":
     version = "1.1"
-    filename = raw_input("Please enter a filename: ")
+    filename = input("Please enter a filename: ")
     write_file(filename, version, "this is test data")
```

```
-    print "File created, reading data from file"
+    print("File created, reading data from file")
     new_version, data = read_file(filename)
-    cents = 73L
+    cents = 73
     quarters = cents / 25
-    print "%s cents contains %s quarters" % (cents, quarters)
+    print("%s cents contains %s quarters" % (cents, quarters))

     new_dict = {}
-    if not new_dict.has_key(new_version):
+    if new_version not in new_dict:
         new_dict[new_version] = filename
```

We can use a patch tool to apply the changes to our source, or we can use `2to3`'s `-w` option to automatically write the changes back to the file while creating a backup copy of the original.

22.4 Testing and common problems

As mentioned previously, there's little chance that the automatic conversions performed by the `2to3` program will be enough for the code to run correctly. In particular, places where the old code handled strings are likely to have problems, because Python 2.x doesn't make a distinction between strings and sequences of raw bytes, and it's next to impossible for any automated conversion tool to consistently make that distinction on its own.

Listing 22.3 is our converted version of listing 22.2, with notes indicating all of the problems that the automatic conversion didn't fix.

Listing 22.3 File convert2_3_converted.py

```
""" example of conversion from Python 2.x to Python 3

"""

def write_file(filename, version, data):
    try:
        outfile = open(filename, "wb")
        outfile.write("%s\n"% (version))          # write needs buffer or bytes, not string
        outfile.write(data)
        outfile.close()

    except Exception as e:
        print(e)

def read_file(filename):
    infile = file(filename, "rb")             # file isn't available; use open
    file_version = infile.readline()
    data = infile.read()
    major_version = int(file_version[0])      # 1
    minor_version = int(file_version[2])
```

```
    if major_version != 1 or minor_version > 5:
        raise Exception("Wrong file version")

    infile.close()
    return file_version, data

if __name__ == "__main__":
    version = "1.1"
    filename = input("Please enter a filename: ")
    write_file(filename, version, "this is test data")
    print("File created, reading data from file")
    new_version, data = read_file(filename)
    cents = 73
    quarters = cents / 25                                    ❷
    print("%s cents contains %s quarters" % (cents, quarters))

    new_dict = {}
    if new_version not in new_dict:
        new_dict[new_version] = filename
```

`file_version` is a `bytes` object, so `int(file_version[0])` yields 49, not 1 ❶. Division returns an int in Python 2.x but a float in Python 3.x, so `quarters` is now 2.92 instead of 2 ❷.

When we attempt to run this file with Python 3, it doesn't run without errors, mostly because of the difference between `bytes` and `strings`. As indicated in the annotations, several errors aren't caught by the conversion. Even when these errors are fixed and the code runs, it doesn't behave correctly, with both the conversion of a single element of a `bytes` object and division behaving differently between the two versions. If these problems occur in such a small script, you can imagine the possibilities for error in a larger application. The moral is pretty clear: if you're migrating code to Python 3.x, you *must* have good test coverage, and you must test extensively.

22.5 Using the same code for 2 and 3

The core developers don't expect that the same code base can be used on both Python 2.x and 3.x. Although Python is flexible enough that in some cases it may be possible to create code that can run on both Python 2.x and 3.x, it isn't recommended.

It's possible to import several features of Python 3, like division, into Python 2.x by using the `__future__` module; but even if you imported all the features of the `__future__` library, the differences in library structure, the distinction between strings and Unicode, and so on all would make for code that would be hard to maintain and debug.

22.5.1 Using Python 2.5 or earlier

If attempting to use the same code for version 3.x and 2.6 and higher isn't recommended, trying to do the same with any earlier version of Python borders on insanity. The differences between earlier versions of Python and Python 3 are large enough

that you'd spend more time making the code run than in designing and implementing an application.

22.5.2 Writing for Python 3.x and converting back

Another way of using one code base for both platforms may be to go in the opposite direction and write code for Python 3.x and convert it *back* to Python 2.x. As this book is being written, a `3to2.py` tool is being developed, with the idea that it will be both easier and more reliable to convert back to 2.x, possibly even as part of the install process, if needed. Although this tool isn't yet complete, it shows great promise for the future.

22.6 Summary

The incompatibility between Python 3 and previous versions is a major consideration when you're dealing with legacy code. Python provides both procedures and tools for moving code from Python 2.x to Python 3, using the warnings available in Python 2.6 and higher and the `2to3` conversion tool, followed by ample testing. Migrating the code Python 3 is probably the ideal long-term solution, but only if the code has adequate test coverage to ensure the new version functions properly.

23 Using Python libraries

This chapter covers

- Managing various data types—strings, numbers, and more
- Manipulating files and storage
- Accessing operating system services
- Using internet protocols and formats
- Developing and debugging tools
- Accessing PyPI, a.k.a. "the Cheese Shop"
- Installing Python libraries using setup.py

Python has long proclaimed that one of its key advantages is its "batteries included" philosophy. This means that a stock install of Python comes with a rich standard library that lets you handle a wide variety of situations without the need to install additional libraries. This chapter will give you a high-level survey of some of the contents of the standard library as well as some suggestions on finding and installing external modules.

23.1 "Batteries included"—the standard library

In Python, what's considered the *library* consists of several different components. It contains built-in data types and constants that can be used without an `import` state-

ment, like numbers and lists, as well as some built-in functions and exceptions. The largest part of the library is an extensive collection of modules. If you have Python, you also have libraries to manipulate diverse types of data and files, to interact with your operating system, to write servers and clients for many internet protocols, and to help develop and debug your code.

What follows is a survey of the high points. Although most of the major modules are mentioned, for the most complete and current information I recommend that you spend some time on your own exploring the library reference that's part of the Python documentation. In particular, before you go in search of an external library, be sure to scan through what Python already offers. You may be surprised at what you find.

23.1.1 Managing various data types

The standard library naturally contains support for Python's built-in types, which we touched on earlier. In addition, three other categories in the standard library deal with various data types: string services, data types, and numeric modules.

String services include the modules in table 23.1 that deal with bytes as well as strings. The three main things these modules deal with are strings and text, sequences of bytes, and Unicode operations.

Table 23.1 String services modules

Module	Description and possible uses
`string`	Compare to string constants, like `digits` or `whitespace`; format strings (see chapter 6)
`re`	Search and replace text using regular expressions (see chapter 17)
`struct`	Interpret bytes as packed binary data; read and write structured data to/from files
`difflib`	Helpers for computing deltas; find differences between strings or sequences, create patches and diff files
`textwrap`	Wrap and fill text; format text by breaking lines or adding spaces

The data types category is a diverse collection of modules covering various data types, particularly time, date, and collections, as shown in table 23.2.

Table 23.2 Data types modules

Module	Description and possible uses
`datetime`, `calendar`	Date, time, and calendar operations
`collections`	Container data types
`array`	Efficient arrays of numeric values

Table 23.2 Data types modules *(continued)*

Module	Description and possible uses
`sched`	Event scheduler
`queue`	Synchronized queue class
`copy`	Shallow and deep copy operations
`pprint`	Data pretty printer

As the name indicates, the numeric and mathematical modules deal with numbers and mathematical operations, and the most common of these are listed in table 23.3. These modules have everything you need to create your own numeric types and handle a wide range of math operations.

Table 23.3 Numeric and mathematical modules

Module	Description and possible uses
`numbers`	Numeric abstract base classes
`math`, `cmath`	Mathematical functions for real and complex numbers
`decimal`	Decimal fixed-point and floating-point arithmetic
`fractions`	Rational numbers
`random`	Generate pseudo-random numbers and choices; shuffle sequences
`itertools`	Functions that create iterators for efficient looping
`functools`	Higher-order functions and operations on callable objects
`operator`	Standard operators as functions

23.1.2 Manipulating files and storage

Another broad category in the standard library covers files, storage, and data persistence and is summarized in table 23.4. It ranges from modules for file access to various modules for data persistence and compression and handling special file formats.

Table 23.4 File and storage modules

Module	Description and possible uses
`os.path`	Common pathname manipulations
`fileinput`	Iterate over lines from multiple input streams
`filecmp`	Compare files and directories
`tempfile`	Generate temporary files and directories

Table 23.4 File and storage modules ***(continued)***

Module	Description and possible uses
`glob`, `fnmatch`	UNIX-style pathname and filename pattern handling
`linecache`	Random access to text lines
`shutil`	High-level file operations
`pickle`, `shelve`	Python object serialization and persistence
`sqlite3`	DB-API 2.0 interface for SQLite databases
`zlib`, `gzip`, `bz2`, `zipfile`, `tarfile`	Work with various archive files and compressions
`csv`	Read and write CSV files
`configparser`	Configuration file parser; read/write Windows-style configuration .ini files

23.1.3 Accessing operating system services

This category is another broad one, containing modules for dealing with your operating system. As shown in table 23.5, this category includes tools for handling command-line parameters, redirecting file and print output and input, writing to log files, running multiple threads or processes, and loading non-Python (usually C) libraries for use in Python.

Table 23.5 Operating system modules

Module	Description
`os`	Miscellaneous operating system interfaces
`io`	Core tools for working with streams
`time`	Time access and conversions
`optparse`	Powerful command-line option parser
`logging`	Logging facility for Python
`getpass`	Portable password input
`curses`	Terminal handling for character-cell displays
`platform`	Access to underlying platform's identifying data
`ctypes`	Foreign function library for Python
`select`	Waiting for I/O completion
`threading`	Higher-level threading interface
`multiprocessing`	Process-based threading interface
`subprocess`	Subprocess management

23.1.4 Using internet protocols and formats

The internet protocols and formats category is concerned with encoding and decoding the many standard formats used for data exchange on the internet, from MIME and other encodings to JSON and XML. It also has modules for writing servers and clients for common services, particularly HTTP, and a generic socket server for writing servers for custom services. The most commonly used are listed in table 23.6.

Table 23.6 Modules supporting internet protocols and formats

Module	Description
`socket`, `ssl`	Low-level networking interface and SSL wrapper for socket objects
`email`	Email and MIME handling package
`json`	JSON encoder and decoder
`mailbox`	Manipulate mailboxes in various formats
`mimetypes`	Map filenames to MIME types
`base64`, `binhex`, `binascii`, `quopri`, `uu`	Encode/decode files or streams with various encodings
`html.parser`, `html.entities`	Parse HTML and XHTML
`xml.parsers.expat`, `xml.dom`, `xml.sax`, `xml.etree.ElementTree`	Various parsers and tools for XML
`cgi`, `cgitb`	Common Gateway Interface support
`wsgiref`	WSGI utilities and reference implementation
`urllib.request`, `urllib.parse`	Open and parse URLs
`ftplib`, `poplib`, `imaplib`, `nntplib`, `smtplib`, `telnetlib`	Clients for various internet protocols
`socketserver`	Framework for network servers
`http.server`	HTTP servers
`xmlrpc.client`, `xmlrpc.server`	XML-RPC client and server

23.1.5 Development and debugging tools and runtime services

Python has several modules to help you debug, test, modify, and otherwise interact with your Python code at runtime. As shown in table 23.7, this includes two different testing tools, profilers, modules to interact with error tracebacks, the interpreter's garbage collection, and so on, as well as modules that let you tweak the importing of other modules.

Table 23.7 Development, debugging, and runtime modules

Module	Description
`pydoc`	Documentation generator and online help system
`doctest`	Test interactive Python examples
`unittest`	Unit testing framework
`test.support`	Utility functions for tests
`pdb`	Python debugger
`profile`, `cProfile`	Python profilers
`timeit`	Measure execution time of small code snippets
`trace`	Trace or track Python statement execution
`sys`	System-specific parameters and functions
`atexit`	Exit handlers
`__future__`	Future statement definitions—features to be added to Python
`gc`	Garbage collector interface
`inspect`	Inspect live objects
`imp`	Access the import internals
`zipimport`	Import modules from zip archives
`modulefinder`	Find modules used by a script

23.2 Moving beyond the standard library

Although Python's "batteries included" philosophy and well-stocked standard library mean that you can do a lot with Python out of the box, there will inevitably come a situation where you need some functionality that doesn't come with Python. This section surveys your options when you need to do something that isn't in the standard library.

23.3 Adding more Python libraries

Finding a Python package or module can be as easy as entering the functionality you're looking for, like "mp3 tags" and "Python" into a search engine, and then sorting through the results. If you're lucky, you may find the module you need packaged for your OS—with an executable Windows or Mac OS X installer or a package for your Linux distribution.

This is one of the easiest ways to add a library to your Python installation, because the installer or your package manager takes care of all the details of adding the module to your system correctly.

In general, such prebuilt packages aren't the rule for Python software. Such packages tend to be a bit older, and there's less flexibility in where and how they're installed.

23.4 Installing Python libraries using setup.py

If you need a third-party module that isn't prepackaged for your platform, you'll have to turn to its source distribution. Installing even a single Python module correctly can involve a certain amount of hassle in dealing with Python's paths and your system's permissions, which makes a standard installation system helpful. The current system uses `distutils`, which is a module in the standard library. Source distributions built to use `distutils` in the standard way should have the name and version number of the package in the archive name, something like spam-0.1.zip, which unpacks into a similarly named directory. After it's unpacked, the distribution contains a setup.py script and a README.txt or README file. That README file includes the instruction that to install the package, you only need to run `python setup.py install` in that directory from a command line.

If you happen to have more than one version of Python installed, it's important to remember that you need to run `setup.py` with the same version of Python that you intend to use with the new package, and you also need to run `setup.py` with sufficient permissions to install software on your system.

For most situations, `python setup.py install` is all you need to know about a `distutils` install. But if you need to install to a different location, possibly because you don't have administrator access on that particular system, you can use the *home* scheme.

23.4.1 Installing under the home scheme

The home scheme is named after the UNIX idea of having a personal home directory for both software and data. In spite of the UNIX-inspired name, this scheme works on any OS.

To install under the home scheme, you need to give the setup.py script the `--home` option followed by the location you want use as the root of your installation. For example, to install into /home/vern, we use

```
python setup.py install --home=/home/vern
```

In this case, after installation, the library modules will be in /home/vern/lib/python, any executable scripts should be in /home/vern/bin, and any data belonging to the package will be in /home/vern/share. To install on a Windows system into C:\Documents and Settings\vern, use the command

```
python setup.py install --home="C:\Documents and Settings\vern"
```

which will install into C:\Documents and Settings\vern\lib, C:\Documents and Settings\vern\bin, and so on. Note that you need to use quotes around any directory names that contain spaces.

As mentioned previously, this scheme is particularly useful if you're working on a system where you don't have sufficient administrator rights to install software, or if you want to install a different version of a module.

23.4.2 Other installation options

`distutils` installations can be tweaked in a wide variety of ways. There are other ways to control the locations installed into, the search path Python uses to locate modules, and so on. If your needs go beyond the basic installation methods discussed here, a good place to start is "Installing Python Modules," which can be found in the Python documentation.

23.5 PyPI, a.k.a. "the Cheese Shop"

Although `distutils` packages get the job done, there's one catch—you have to find the correct package, which can be a chore. And after you've found a package, it would be nice to have a reasonably reliable source from where to download that package.

To meet this need, there have been different Python package repositories over the years, and currently the official (but by no means the only) repository for Python code is the Python Package Index, or PyPI (also known as "the Cheese Shop") on the Python website. You can access it from a link on the main page or directly at http://pypi.python.org. PyPI contains over 6,000 packages for various Python versions, listed by date added and name, but also searchable and broken down by category.

PyPI is the logical next stop if you can't find the functionality you want with a search of the standard library.

23.6 Summary

Python has a rich standard library that covers more common situations than many other languages, and you should check what's in the standard library carefully before looking for external modules. If you do need an external module, prebuilt packages for your OS are the easiest option, but they're sometimes older and often hard to find. If you can't find a prebuilt package, the standard way to install from source is using setup.py from the `distutils` package. In any case, the logical first step in searching for external modules is the Python Package Index, or PyPI.

24 Network, web, and database programming

This chapter covers

- Accessing databases in Python
- Network programming in Python
- Creating Python web applications
- Writing a sample project: creating a message wall

By this point, we've surveyed a lot of what Python can do. The final area to examine is one of the most important: using Python to build web applications that serve dynamic content. Many libraries and frameworks for creating web applications are available, in many languages—Java, PHP, Perl, and Ruby to name a few. As web applications continue to evolve and increase in importance, Python's role in this space continues to grow.

Dynamic web applications typically store their content in databases and use the results of queries on those databases to generate page content dynamically. Various templating libraries and application frameworks are commonly used to handle URLs and format content. In this chapter, we'll look at the pieces of this process

using simple examples. When you see how the pieces fit together in Python, using almost any combination of database and application framework is possible.

24.1 Accessing databases in Python

Database access is a large part of many applications, including dynamic web applications. By using external modules, Python can access most popular databases, and in general the interface for each follows the DB-API 2.0 standard database specification detailed in PEP (Python Enhancement Proposal) 249. The specification calls for the use of a `connection` object to manage the connection to the database and for the use of `cursor` objects to manage the interaction with the database, for fetching data from the database and updating its contents.

The fact that Python database libraries conform to the DB-API 2.0 spec has a couple of obvious advantages. For one thing, writing code for different databases is easier, because the general rules are the same. The other advantage is that it's fairly easy to prototype an application using a lightweight database and then switch the application over to a production database after the basic design of the application has been finalized.

24.1.1 Using the sqlite3 database

Although there are Python modules for many databases, for the following examples we'll look at the one that comes included with Python: sqlite3. Although not suited for large, high-traffic applications, sqlite3 has two advantages: first, because it's part of the standard library it can be used anywhere you need a database, without worrying about adding dependencies; second, sqlite3 stores all of its records in a local file, so it doesn't need both a client and server, like MySQL or other common databases. These features make sqlite3 a handy option for both smaller applications and quick prototypes.

To use a sqlite3 database, the first thing you need is a `connection` object. Getting a `connection` object requires only calling the `connect` function with the name of file that will be used to store the data:

```
>>> import sqlite3
>>> conn = sqlite3.connect("datafile")
```

It's also possible to hold the data in memory by using `":memory:"` as the filename. For storing Python integers, strings, and floats, nothing more is needed. If you want sqlite3 to automatically convert query results for some columns into other types, it's useful to include the `detect_types` parameter set to `sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES`, which will direct the `connection` object to parse the name and types of columns in queries and attempt to match them with converters you've already defined.

The second step is to create a `cursor` object from the connection:

```
>>> cursor = conn.cursor()
>>> cursor
<sqlite3.Cursor object at 0xb7a12980>
```

At this point, you're able to make queries against the database. In our current situation, because there are no tables or records in the database yet, we need to create one and insert a couple of records:

```
>>> cursor.execute("create table test (name text, count integer)")
>>> cursor.execute("insert into test (name, count) values ('Bob', 1)")
>>> cursor.execute("insert into test (name, count) values (?, ?)",
...                ("Jill", 15))
```

The last `insert` query illustrates the preferred way to make a query with variables; rather than constructing the query string, it's more secure to use a `?` for each variable and then pass the variables as a tuple parameter to the `execute` method. The advantage is that you don't need to worry about incorrectly escaping a value; sqlite3 takes care of it for you.

You can also use variable names prefixed with `:` in the query and pass in a corresponding dictionary with the values to be inserted:

```
>>> cursor.execute("insert into test (name, count) values (:username, \
                   :usercount)", {"username": "Joe", "usercount": 10})
```

After a table is populated, you can query the data using SQL commands, again using either `?` for variable binding or names and dictionaries:

```
>>> result = cursor.execute("select * from test")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 15), ('Joe', 10)]
>>> result = cursor.execute("select * from test where name like :name",
...                         {"name": "bob"})
>>> print(result.fetchall())
[('Bob', 1)]
>>> cursor.execute("update test set count=? where name=?", (20, "Jill"))
>>> result = cursor.execute("select * from test")
>>> print(result.fetchall())
[('Bob', 1), ('Jill', 20), ('Joe', 10)]
```

In addition to the `fetchall` method, the `fetchone` method gets one row of the result and `fetchmany` returns an arbitrary number of rows. For convenience, it's also possible to iterate over a cursor object's rows similar to iterating over a file:

```
>>> result = cursor.execute("select * from test")
>>> for row in result:
...     print(row)
...
('Bob', 1)
('Jill', 20)
('Joe', 10)
```

Finally, by default, sqlite3 doesn't immediately commit transactions. That means you have the option of rolling back a transaction if it fails, but it also means you need to use the `connection` object's `commit` method to ensure that any changes made have been saved. This is a particularly good idea before you close a connection to a database, because the `close` method doesn't automatically commit any active transactions:

```
>>> cursor.execute("update test set count=? where name=?", (20, "Jill"))
>>> conn.commit()
>>> conn.close()
```

Table 24.1 gives an overview of the most common operations on a sqlite3 database.

Table 24.1 Common database operations

Operation	Sqlite3 command
Create a connection to a database	`conn = sqlite3.connect(filename)`
Create a cursor for a connection	`Cursor = conn.cursor()`
Execute a query with the cursor	`cursor.execute(query)`

Return the results of a query	`cursor.fetchall(),cursor.fetchmany(num_rows), cursor.fetchone()`
	`for row in cursor:`
	`    ....`

Commit a transaction to a database	`conn.commit()`
Close a connection	`conn.close()`

These operations are usually all you need to manipulate a sqlite3 database. Of course, several options let you control their precise behavior; see the Python documentation for more information.

24.2 Network programming in Python

The Python standard library has everything you need to handle the standard internet protocols and to create both clients and servers. The following examples use HTTP, but similar patterns are used in most protocols.

24.2.1 Creating an instant HTTP server

The standard library has a number of modules for writing servers for various network protocols. In many cases, you can create a server in only a few lines of code. Suppose we want to make a particular folder's files freely accessible via HTTP, perhaps to share a few files with coworkers without the hassle of setting up a formal repository or file share. With Python, we don't need to install and configure a server. With a few lines of code, we can create a temporary server on the fly:

```
>>> from http.server import HTTPServer, SimpleHTTPRequestHandler
>>> server = HTTPServer(("", 8000), SimpleHTTPRequestHandler)
>>> server.serve_forever()
```

This server will serve the contents of the folder it's run in on port 8000 for all active network interfaces. The tuple `("", 8000)` sets the address and port for the server. Leaving the address an empty string allows it to use all of the machine's network addresses, and the `8000` specifies the port. The second parameter to `HTTPServer` is the

request handler; and `SimpleHTTPRequestHandler` is a subclass of `BaseHTTPRequestHandler`, which is written to serve files from the current directory. In particular, `SimpleHTTPRequestHandler` defines `do_GET` and `do_HEAD` methods, which map the requests to the contents of the current folder, returning either directory listings or the contents of a file, as needed.

Another predefined request-handler class, `CGIHTTPRequestHandler`, serves files or the output of CGI scripts in the current folder. `CGIHTTPRequestHandler` defines an additional method, `do_POST`, which responds to POST requests on CGI scripts. Any of the request-handler classes can be subclassed to create the particular behavior you desire.

24.2.2 Writing an HTTP client

Writing Python code to interact with an HTTP server is also quite easy. The `urllib.request` module is designed to enable interaction with URLs in all their real-world complexity, including authentication, cookies, redirections, and the like.

But for all of its power and options, basic interactions with the `urllib.request` module are simple. The URL is opened and returns a file-like object, which you can read and treat like any other file. The object returned also has two additional methods: `geturl`, which returns the URL retrieved, so you can tell if the original request was redirected; and `info`, which returns the page's headers. For example, if we were to leave our simple HTTP server running and open another Python interactive shell, we could access the server in this way:

```
>>> from urllib.request import urlopen
>>> url_file = urlopen("http://localhost:8000")
>>> print(url_file.geturl())
http://localhost:8000
>>> print(url_file.info())
Server: SimpleHTTP/0.6 Python/3.1.1
Date: Sat, 06 Jun 2009 20:28:13 GMT
Content-type: text/html; charset=utf-8
Content-Length: 15395


>>> for line in url_file.readlines():
...     print(line)
...
b'<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>\n'
b'<title>Directory listing for /</title>\n'
b'<body>\n'
b'<h2>Directory listing for /</h2>\n'
b'<hr>\n'
b'<ul>\n'
...
# file names would be in HTML links in list items here...
...
b'</ul>\n'
b'<hr>\n'
b'</body>\n'
b'</html>\n'
```

24.3 Creating a Python web application

Although the `http.server` module has the basics of a web server, you need more than the basics to write a full-featured web application. You need to manage users, authentication, and sessions; you need a way to generate HTML pages. The solution to this is to use a web framework, and over the years many frameworks have been created in Python, leading to the present generation, which includes Zope and Plone, Django, TurboGears, web2py, and many more.

One thing to keep in mind in considering web applications is that the *web server* functionality—the part that processes HTTP requests and returns responses—isn't necessarily (or even usually) closely tied to the web application itself. Most web frameworks today can either run their own HTTP servers or rely on an external server, depending on the situation. For development, it's often preferred to use the application's internal server, because there's less to install and set up. For heavy traffic in production, often an external server handles the load better and can be tweaked as needed.

24.3.1 Using the web server gateway interface

In the early days of Python web frameworks, there was little standardization of how web applications interacted with web servers. Consequently, the choice of a web framework often limited the possible web servers you could use and made migration from one to the other difficult. The WSGI ("whiz-ghee"), or web server gateway interface, specification was created to provide a standard for the interaction of web server and applications, so that it would be easier to use a web application and framework on different web servers. Figure 24.1 is a simple schematic of how a WSGI application combines with a server.

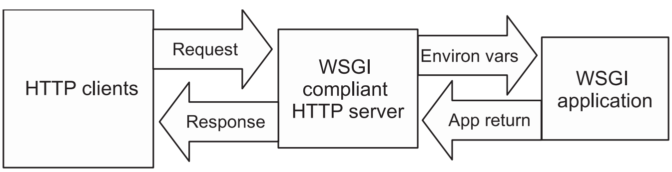


Figure 24.1 How a WSGI application works with a server

You need to know the details of WSGI design only if you're writing web servers or web application frameworks, but it's useful to understand the basic idea of how a WSGI application is set up.

24.3.2 Using the wsgi library to create a basic web app

Although WSGI is technically more of a specification than a library, the standard library does include a reference implementation of WSGI server and utilities in the `wsgiref` module. It includes a simple server implementation as well as various utilities. Creating a basic WSGI application using `wsgiref.simple_server` is almost as easy as our bare-bones HTTP server earlier. The one thing you need to pass to the server when it's created is a callable application object that receives two parameters from the server: a dictionary of environment variables and a callable object to receive HTTP status and

headers for the response. Here's a basic "Hello World" WSGI application, based on the one in the standard library documentation:

```
from wsgiref.simple_server import make_server

def hello_world_app(environ, start_response):
    status = b'200 OK'                                          # HTTP status
    headers = [(b'Content-type', b'text/plain; charset=utf-8')]
    start_response(status, headers)

    return [b"Hello World"]                      # Returned object will be printed

httpd = make_server('', 8000, hello_world_app)
print("Serving on port 8000...")

httpd.serve_forever()                            # Serves until process is killed
```

If you run this script and open a browser to http://127.0.0.1:8000, you should be greeted by "Hello World."

The two main parts of this are `hello_world_app` and the creation of the server itself. The creation of the server is easy: a call to `make_server:make_server`, passing in the application object, and the address and port for the server. In this example the address is blank, so the server will listen on all active interfaces. After the server has been created, you start it by calling its `serve_forever:serve_forever` method; or you can make it serve only one request and quit by using the `handle_request:handle_request` method.

Creating the application object, `hello_world_app`, is more involved but still fairly simple, because it's a function with the two required parameters, `environ` and `start_response`. The former is a CGI-style dictionary of variables from the server, which in this example is ignored. The second parameter, `start_response`, is a callable object that the application uses to send the HTTP status and headers of its response. In this case, as usual, we send a status of `'200 OK'` and the headers indicating that this response will be plain text, encoded as UTF-8. We'll be using this basic implementation as the starting point to create a simple "message wall" application at the end of this chapter.

24.3.3 Using frameworks to create advanced web apps

The WSGI specification goes a long way toward standardizing the way that Python web applications can interact with web servers and handle HTTP requests and responses. It doesn't specify how applications behave internally, however, nor does it mandate what the application does with the requests it receives or how it generates its responses. This means that with a bare WSGI application a lot must be implemented. And a lot of that is repetitive: handling URLs, extracting the information from web forms, creating HTML pages, inserting dynamic content into those pages, and the like. Although it's possible to code everything by hand or to assemble and/or write a custom collection of modules to handle these chores, for most projects it's preferable to use an existing

web framework. The most popular frameworks include a server (good enough for at least development and testing) and the machinery for creating and customizing web applications.

At the time of this writing, none of the major frameworks have been ported to Python 3, but the work is in progress; Python 3 versions of all the major web app frameworks should be ready by the time Python 3.2 is released in the summer of 2010.

24.4 Sample project—creating a message wall

To see what you can do with a WSGI application, let's create a simple example: a message wall where the messages are stored in a sqlite3 database and the URL is used to indicate the user and tags being searched for. Note that this is a simple example, and we won't be implementing any kind of security, session management, and so on, so don't use this in production. This application is intended to show you the possibilities for writing web applications in Python.

As you've seen, we need to create an application object to pass to the server when we create it. This application needs to be able to do the following:

- Retrieve messages written by a user, and display them along with their ages: for example, "5 minutes ago," "2 days ago," and so on
- Retrieve messages addressed to a user (messages beginning with `@user`), and display them and their ages
- Display messages in chronological order from newest to oldest
- Allow users to enter messages and store them in the database

24.4.1 Creating the database

The first thing we need to do is to create the sqlite3 database for the app. We'll need three fields in the messages table: the user, the message itself, and its timestamp. Both the user and the message are naturally text fields in sqlite3 and will map automatically to strings in Python. For the timestamp, it would be convenient if we could read the values as Python `datetime` values. Fortunately, Python's sqlite3 module includes converters for date and `datetime` types; we'll need to activate type detection on the `connection` object in order to have a timestamp database type available. To set up the database, we need to enter the following from a Python shell in the directory where we want to run our database:

```
>>> import sqlite3
>>> conn = sqlite3.connect("messagefile", \
...        detect_types=sqlite3.PARSE_DECLTYPES | sqlite3.PARSE_COLNAMES)
>>> cursor = conn.cursor()
>>> cursor.execute("create table messages(user text, message text, "
...                "ts timestamp)")
>>> conn.commit()
>>> conn.close()
```

This creates a database in the file messagefile and creates a table called messages with text fields for the user and the message and a timestamp field called ts.

24.4.2 Creating an application object

The next step is to create the application. First, let's write the application to display the title Message Wall, as shown in listing 24.1.

Listing 24.1 File message_wall01.py

```
from wsgiref.simple_server import make_server

def message_wall_app(environ, start_response):
    status = b'200 OK'                                          <-- HTTP status
    headers = [(b'Content-type', b'text/html; charset=utf-8')]
    start_response(status, headers)

    return ["<h1>Message Wall</h1>"]                            <-- Returned object will be printed

httpd = make_server('', 8000, message_wall_app)
print("Serving on port 8000...")

httpd.serve_forever()                                           <-- Serves until process is killed
```

Running this program gets a similar result to our early simple example. When you visit 127.0.0.1:8000 with a web browser, you see only "Message Wall." Note, however, that we did change the type of the response in our header from `text/plain` to `text/html`.

24.4.3 Adding a form and retrieving its contents

With the basic application working, it's time to add its functionality. The first thing we can do is add a form for message submissions. This will require two things: sending the HTML code to make the form and then retrieving its values when the user submits the form. Sending the HTML code means that we'll be creating longer and longer strings to return from our application object; rather than using string concatenation, we can use a `StringIO` object from the io library to let us print our output as if we were writing it to a file and then return it as one string.

Retrieving the values of the form can happen only when values are submitted, which makes the request's `REQUEST_METHOD` be `POST` instead of the normal `GET`. When that happens, we can get the values from the form by using the `CONTENT_LENGTH` item from `environ` to find out how long the form string is and then reading that amount from `environ`'s `wsgi.input`. At this point, we aren't saving the form data, but we'll print it to the browser so that we can see what's going on (see listing 24.2).

Listing 24.2 File message_wall02.py

```
from wsgiref.simple_server import make_server
from io import StringIO

def message_wall_app(environ, start_response):
    output = StringIO()
    status = b'200 OK' # HTTP Status
    headers = [(b'Content-type', b'text/html; charset=utf-8')]
    start_response(status, headers)
```

```
    print("<h1>Message Wall</h1>", file=output)
    if environ['REQUEST_METHOD'] == 'POST':
        size = int(environ['CONTENT_LENGTH'])
        post_str = environ['wsgi.input'].read(size)
        print(post_str, "<p>", file=output)
    print('<form method="POST">User: <input type="text" '
          'name="user">Message: <input type="text" '
          'name="message"><input type="submit" value="Send"></form>',
           file=output)
    return [output.getvalue()]

httpd = make_server('', 8000, message_wall_app)
print("Serving on port 8000...")

httpd.serve_forever()
```

Checks for `POST` instead of `GET`

❶ ❷

We use the `print` function's `file` parameter to print to `StringIO` object ❶. We then return the entire contents of `output` by calling its `getvalue` method ❷.

Figure 24.2 is a screen shot of the browser window, with the server running in a command window behind it.

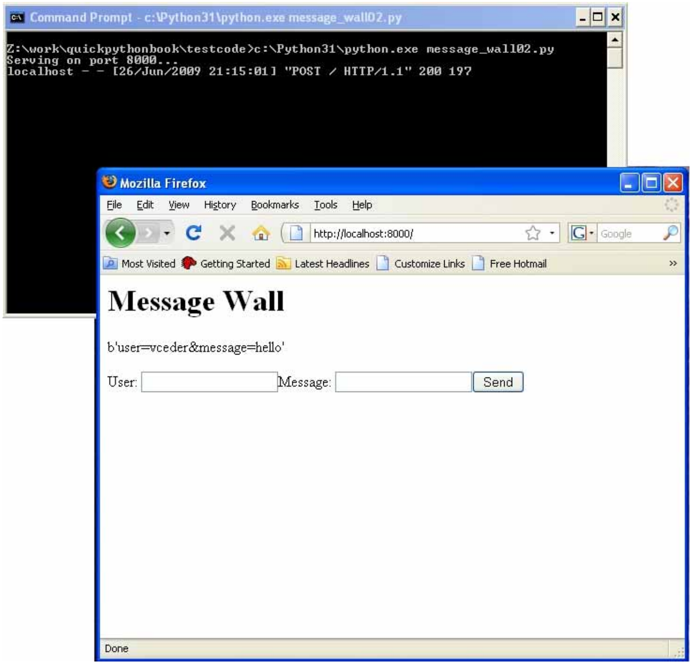


Figure 24.2 The browser and server after a form `POST`

Note that the value of the form fields is returned as a single `bytes` object.

24.4.4 Saving the form's contents

When we have a message, we need to save the message to the database. To do this, we'll need to parse the value we read from the form and then use a sqlite3 query to store it in the database. To make this easier, we've added a helper function to parse

the form string into a dictionary, and we've added the current timestamp to that dictionary. Then, the dictionary is used as the parameter for a SQL query to save the message to the database (see listing 24.3).

Listing 24.3 File message_wall03.py

```
from wsgiref.simple_server import make_server
from io import StringIO
import sqlite3
import datetime

def get_form_vals(post_str):
    form_vals = {item.split("=")[0]: item.split("=")[1] for item
                 in post_str.decode().split("&")}              ❶
    return form_vals

def message_wall_app(environ, start_response):
    output = StringIO()
    status = b'200 OK' # HTTP Status
    headers = [(b'Content-type', b'text/html; charset=utf-8')]
    start_response(status, headers)
    print("<h1>Message Wall</h1>", file=output)
    if environ['REQUEST_METHOD'] == 'POST':
        size = int(environ['CONTENT_LENGTH'])
        post_str = environ['wsgi.input'].read(size)
        form_vals = get_form_vals(post_str)
        form_vals['timestamp'] = datetime.datetime.now()       ❷
        print(form_vals, "<p>", file=output)
        cursor.execute("insert into messages (user, message, ts) values "
                       "(:user, :message, :timestamp)", form_vals)
    print('<form method="POST">User: <input type="text" '
          'name="user">Message: <input type="text" '
          'name="message"><input type="submit" value="Send"></form>',
           file=output)
    return [output.getvalue()]

httpd = make_server('', 8000, message_wall_app)
print("Serving on port 8000...")

conn = sqlite3.connect("messagefile")                          ❸
cursor = conn.cursor()
httpd.serve_forever()
```

We use a dictionary comprehension to transform `post_str` into a dictionary ❶. Then, we add a timestamp to the `form_vals` dictionary ❷. Sqlite3 connection and cursor objects are created only once for an application ❸.

24.4.5 Parsing the URL and retrieving messages

What remains is to retrieve messages from or addressed to a particular user. As described in our requirements, a message addressed to a user will begin with `@` and that user's username. Also shown in the specs, the desired way of indicating the user is to append the username to the base URL. To get all messages to user `vceder`, for

example, that username would be added to the URL, which in the case of our localhost server would be http://localhost:8000/vceder.

That means we need to get the PATH_INFO:PATH_INFO from the request, which is anything beyond the base URL, and split it on/into fields. We can use Python's flexible tuple unpacking to put the first field into a username variable and anything beyond the first field into a tags list, so that we can easily add tagging later.

In this case, it's simpler to create a query using ? to mark the variables. We can create a tuple of (user, "@"+user+"%") to hold the values, with the first part of the query looking for an exact match on the username and the second part being a match with any message that starts with an @ and the username.

We've also taken care of a couple of other issues. First, a helper function message_table puts any matching records from the query into an HTML table. Second, the post_str variable is now decoded from a bytes object to a string, and it also has any quoting removed with the unquote_plus function from the urllib.parse library (see listing 24.4).

Listing 24.4 File message_wall04.py

```
from wsgiref.simple_server import make_server
from io import StringIO
from urllib.parse import unquote_plus
import sqlite3
import datetime

def get_form_vals(post_str):
    form_vals = {item.split("=")[0]: item.split("=")[1] for item \
                 in post_str.split("&")}
    return form_vals

def message_table(messages):
    table = "<table>\n"
    for message in messages:
       row_str = "<tr><td>{0}</td><td>{1}</td><td>{2}</td></tr>\n"
       table += row_str.format(message[2], message[0], message[1])
    table += "</table>"
    return table

def message_wall_app(environ, start_response):
    from io import StringIO
    output = StringIO()
    status = b'200 OK' # HTTP Status
    headers = [(b'Content-type', b'text/html; charset=utf-8')]
    start_response(status, headers)
    print("<h1>Message Wall</h1>", file=output)
    if environ['REQUEST_METHOD'] == 'POST':
       size = int(environ['CONTENT_LENGTH'])
       post_str = environ['wsgi.input'].read(size)
       post_str = unquote_plus(post_str.decode())      ❶
       form_vals = get_form_vals(post_str)
       form_vals['timestamp'] = datetime.datetime.now()
```

```
        cursor.execute("""insert into messages (user, message, ts) values
                       (:user, :message, :timestamp)""", form_vals)
    path_vals = environ['PATH_INFO'][1:].split("/")                    ❷
    user,*tag = path_vals
    cursor.execute("""select * from messages where user like ? or message
                    like ? order by ts""", (user, "@" + user + "%"))   ❸
    print(message_table(cursor.fetchall()), "<p>", file=output)


    print('<form method="POST">User: <input type="text" '
          'name="user">Message: <input type="text" '
          'name="message"><input type="submit" value="Send"></form>',
           file=output)
    return [output.getvalue()]

httpd = make_server('', 8000, message_wall_app)
print("Serving on port 8000...")

conn = sqlite3.connect("messagefile")
cursor = conn.cursor()
httpd.serve_forever()
```

Here we decode from bytes to string and unquote ❶. We read and parse `PATH_INFO` for username ❷. Then, we query the database for matching messages ❸.

Figure 24.3 is a screenshot of the server and browser after a message has been posted.

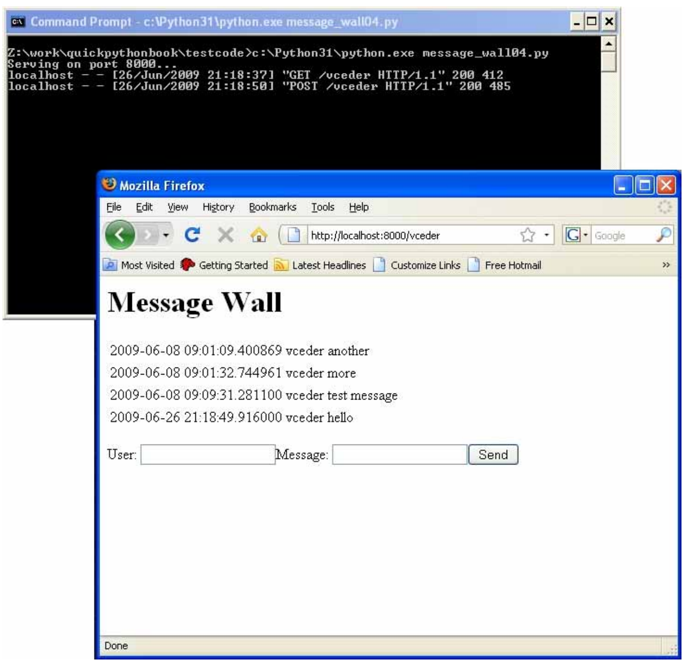


Figure 24.3 Browser and server after a message `POST`

If you compare the timestamp of the message in the browser and the time of the POST statement in the server's command window, you can see that the last message has just been posted.

24.4.6 Adding an HTML wrapper

If we were to use our browser's View Source feature to look at the HTML this application is generating, we'd see that we're not returning complete pages. Instead, we're returning only the HTML code for the contents of the page, with no beginning or end. The last step is to wrap our output in HTML headers and footers to make it a more legal HTML page. To do that, we can add some variables to hold header and footer strings and a little function to put everything together (listing 24.5).

Listing 24.5 File message_wall05.py

```
from wsgiref.simple_server import make_server
from io import StringIO
from urllib.parse import unquote_plus
import sqlite3
import datetime

header = "<html><header> <title>Message Wall</title></header><body>"
footer = "</body></html>"

def html_page(content):
    page = "%s\n%s\n%s" % (header, content, footer)
    return page

def get_form_vals(post_str):
    form_vals = {item.split("=")[0]: item.split("=")[1] for item \
                 in post_str.split("&")}
    return form_vals

def message_table(messages):
    table = "<table>\n"
    for message in messages:
        row_str = "<tr><td>{0}</td><td>{1}</td><td>{2}</td></tr>\n"
        table += row_str.format(message[2], message[0], message[1])
    table += "</table>"
    return table

def message_wall_app(environ, start_response):
    from io import StringIO
    output = StringIO()
    status = b'200 OK' # HTTP Status
    headers = [(b'Content-type', b'text/html; charset=utf-8')]
    start_response(status, headers)
    print("<h1>Message Wall</h1>", file=output)
    if environ['REQUEST_METHOD'] == 'POST':
        size = int(environ['CONTENT_LENGTH'])
        post_str = unquote_plus(environ['wsgi.input'].read(size).decode())
```

```
        form_vals = get_form_vals(post_str)
        form_vals['timestamp'] = datetime.datetime.now()
        cursor.execute("""insert into messages (user, message, ts) values
                       (:user, :message, :timestamp)""", form_vals)
    path_vals = environ['PATH_INFO'][1:].split("/")
    user, *tag = path_vals
    cursor.execute("""select * from messages where user like ? or message
                    like ? order by ts""", (user, "@" + user + "%"))
    print(message_table(cursor.fetchall()), "<p>", file=output)

    print('<form method="POST">User: <input type="text" '
          'name="user">Message: <input type="text" '
          'name="message"><input type="submit" value="Send"></form>',
          file=output)
    return [html_page(output.getvalue())]

httpd = make_server('', 8000, message_wall_app)
print("Serving on port 8000...")

conn = sqlite3.connect("messaging")
cursor = conn.cursor()
httpd.serve_forever()
```

Although it works fine for a tiny demo application, this method of applying HTML formatting can become tedious as pages and their contents get more complex. In addition, hand-coding the generation of HTML can make designing the appearance of the page doubly burdensome, because whoever is doing it must be both a programmer and a designer. Obviously, this example suffers from a plain look that could be dressed up with advanced HTML code. Web application frameworks use some sort of templating system to handle formatting so that you don't need to worry about it.

24.5 Summary

The previous example illustrates the basics of how Python combines databases and the WSGI standard to create dynamic web applications. The choices of web application frameworks in Python are extensive, but almost all of them work in the same way and handle the same basic problems: handling requests, parsing URLs and mapping them to parts of the web application, handling user data, creating dynamic pages based on database queries, and producing HTML.

appendix

A guide to Python's documentation

The best and most current reference for Python is the documentation that comes with Python itself. With that in mind, it will be more useful to explore the ways you can access that documentation than to print pages of edited documentation.

The standard bundle of documentation has several sections, including instructions on documenting, distributing, installing, and extending Python on various platforms, and is the logical starting point when you're looking for answers to questions about Python. The two main areas of the Python documentation that are likely to be the most useful are the *Library Reference* and the *Language Reference*. The *Library Reference* is absolutely essential, because it has explanations of both the built-in data types and every module included with Python. The *Language Reference* is the explanation of how the core of Python works, and it contains the official word on the core of the language, explaining the workings of data types, statements, and so on. The "What's New" section is also worth reading, particularly when a new version of Python is released, because it summarizes all of the changes in the new version.

Accessing Python documentation on the Web

For many people, the most convenient way to access the Python documentation is to go to www.python.org and browse the documentation collection there. Although this requires a connection to the web, it has the advantage that the content is always the most current. Given that for many projects it's often useful to search the web for other documentation and information, having a browser tab permanently open and pointing to the online Python documentation is an easy way to have a Python reference at your fingertips.

Browsing Python documentation on your computer

Many distributions of Python include the full documentation by default. In some Linux distributions, the documentation is a separate package that you need to install separately. In most cases, however, full documentation is already on your computer and easily accessible.

Accessing help in the interactive shell or at a command line

In chapter 2, you saw how to use the `help` command in the interactive interpreter to access online help for any Python module or object:

```
>>> help(int)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.  A floating
 |  point argument will be truncated towards zero (this does not include a
 |  string representation of a floating point number!)  When converting a
 |  string, use the optional base.  It is an error to supply a base when
 |  converting a non-string.
 |
 |  Methods defined here:
... (continues with a list of methods for an int)
```

What's happening is that the interpreter is calling the `pydoc` module to generate the documentation. You can also use the `pydoc` module to search the Python documentation from a command line. On a Linux or Mac OS X system, to get the same output in a terminal window, you need only type `pydoc int` at the prompt; to exit, type `q`. In a Windows command window, unless you've set your search path to include the Python Lib directory, you'll need to type the entire path, something like `C:\Python31\Lib\pydoc.py int`.

Generating HTML help pages with pydoc

If you want a sleeker look to the documentation that `pydoc` generates for a Python object or module, you can also have the output written to an HTML file, which you can view in any browser. To do this, add the `-w` option to the `pydoc` command, which on Windows would then be `C:\Python31\Lib\pydoc.py -w int`. In this case, where we're looking up documentation on the `int` object, `pydoc` will create a file named int.html in the current directory, and we can open and view it in a browser from there. Figure A.1 shows what int.html looks like in a browser.

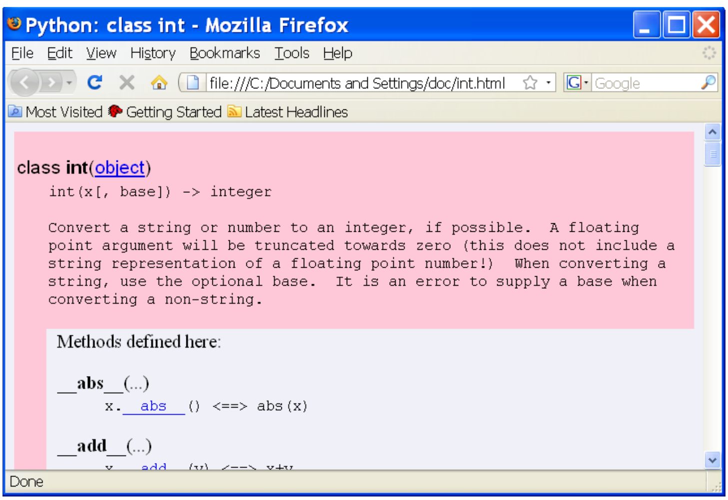


Figure A.1 int.html as generated by `pydoc`

If for some reason you want only a limited number pages of documentation available, this method works well. But in most cases it will probably be better to use `pydoc` to serve more complete documentation, as discussed in the next section.

USING PYDOC AS A DOCUMENTATION SERVER

In addition to being able to generate text and HTML documentation on any Python object, the `pydoc` module can also be used as a server to serve web-based docs. You can do this two ways. Either you can run `pydoc` with `-p` and a port number to open a server on that port, or you can run `pydoc -g` to pop up a little search dialog box and open a browser from there, as illustrated in figure A.2.

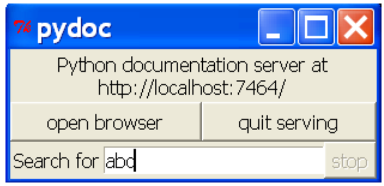


Figure A.2 `pydoc` search dialog box

Clicking the Open Browser button opens your system's default browser and gives you access to the documentation of all the modules available, as shown in figure A.3.

A bonus in using `pydoc` to serve documentation is that it also scans the current directory and extracts documentation from the docstrings of any modules it finds, even if they aren't part of the standard library. This makes it useful for accessing the documentation of any Python modules. There is one caveat, however. To extract the

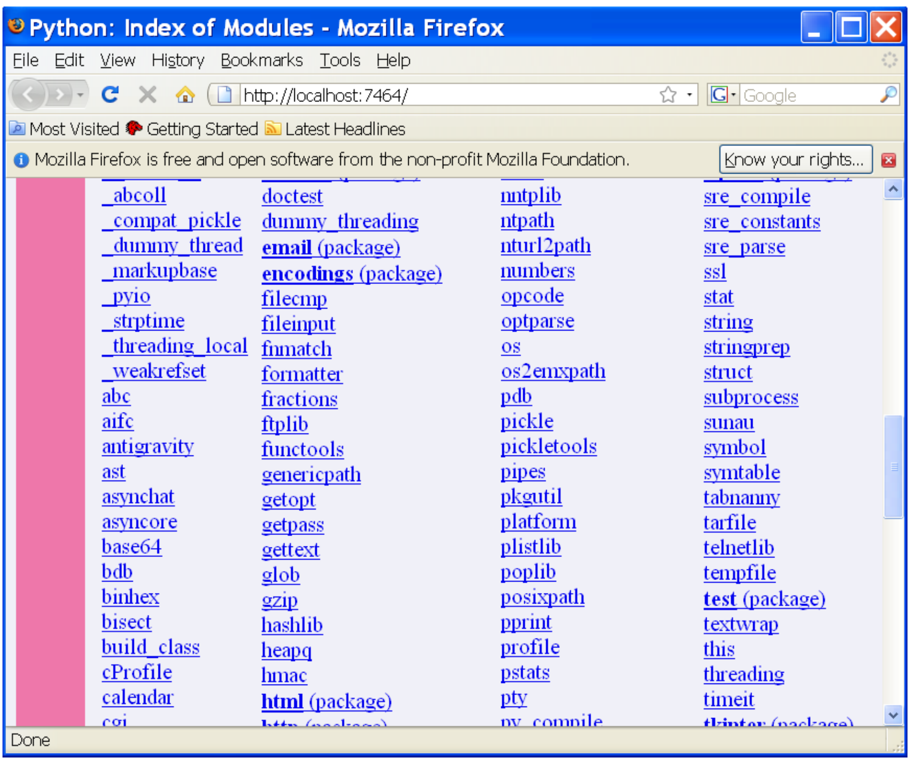


Figure A.3 A partial view of the module documentation served by `pydoc`



Figure A.4 Python documentation in a Windows Help file

documentation from a module, pydoc must import it, which means it will execute any code at the module's top level. Thus scripts that aren't written to be imported without side effects, as discussed in chapter 11, will be run, so use this feature with care.

USING THE WINDOWS HELP FILE

On Windows systems, the standard Python 3 package includes complete Python documentation as a Windows Help file. You can find it in the Doc folder inside the folder where Python was installed (C:\Python31\Doc on my system). When you open it, it will look something like figure A.4.

If you're comfortable with using Window Help files, this file may be all the documentation you ever need.

Downloading documentation

If you want the Python documentation on a computer but don't necessarily want or need to be running Python, you can also download the complete documentation from python.org in PDF, HTML, or text format. This is convenient if you want to be able to access the docs from an ebook reader or similar device.

The Python manual of style

This section contains a slightly edited excerpt from PEP (Python Enhancement Proposal) 8. Written by Guido van Rossum and Barry Warsaw, PEP 8 is the closest thing Python has to a style manual. Some more-specific sections have been omitted, but the

main points are covered. You should make your code conform to PEP 8 as much as possible—your Python style will be the better for it.

You can access the full text of PEP 8 and all of the other PEPs issued in the history of Python by going to www.python.org's documentation section and looking for the PEP index. The PEPs are an excellent source for the history and lore of Python as well as explanations of current issues and future plans.

PEP 8 - Style Guide for Python Code

Introduction

This document gives coding conventions for the Python code comprising the standard library in the main Python distribution. Please see the companion informational PEP describing style guidelines for the C code in the C implementation of Python.[1] This document was adapted from Guido's original Python Style Guide essay,[2] with some additions from Barry's style guide.[3] Where there's conflict, Guido's style rules for the purposes of this PEP. This PEP may still be incomplete (in fact, it may never be finished <wink>).

A FOOLISH CONSISTENCY IS THE HOBGOBLIN OF LITTLE MINDS

One of Guido's key insights is that code is read much more often than it's written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20[4] says, "Readability counts."

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.

But most important, know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!

Here are two good reasons to break a particular rule:

- When applying the rule would make the code less readable, even for someone who is used to reading code that follows the rules
- To be consistent with surrounding code that also breaks it (maybe for historic reasons), although this is also an opportunity to clean up someone else's mess (in true XP style)

Code layout

INDENTATION

Use four spaces per indentation level.

1 PEP 7, Style Guide for C Code, van Rossum
2 http://www.python.org/doc/essays/styleguide.html
3 Barry's GNU Mailman style guide: http://barry.warsaw.us/software/STYLEGUIDE.txt
4 PEP 20, The Zen of Python

For really old code that you don't want to mess up, you can continue to use eight-space tabs.

Tabs or spaces?

Never mix tabs and spaces.

The most popular way of indenting Python is with spaces only. The second most popular way is with tabs only. Code indented with a mixture of tabs and spaces should be converted to using spaces exclusively. When you invoke the Python command-line interpreter with the `-t` option, it issues warnings about code that illegally mixes tabs and spaces. When you use `-tt`, these warnings become errors. These options are highly recommended!

For new projects, spaces only are strongly recommended over tabs. Most editors have features that make this easy to do.

Maximum line length

Limit all lines to a maximum of 79 characters.

Many devices are still around that are limited to 80-character lines; plus, limiting windows to 80 characters makes it possible to have several windows side by side. The default wrapping on such devices disrupts the visual structure of the code, making it more difficult to understand. Therefore, please limit all lines to a maximum of 79 characters. For flowing long blocks of text (docstrings or comments), limiting the length to 72 characters is recommended.

The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets, and braces. If necessary, you can add an extra pair of parentheses around an expression, but sometimes using a backslash looks better. Make sure to indent the continued line appropriately. The preferred place to break around a binary operator is *after* the operator, not before it. Here are some examples:

```
class Rectangle(Blob):
    def __init__(self, width, height,
                 color='black', emphasis=None, highlight=0):
        if width == 0 and height == 0 and \
           color == 'red' and emphasis == 'strong' or \
           highlight > 100:
            raise ValueError("sorry, you lose")
        if width == 0 and height == 0 and (color == 'red' or
                                           emphasis is None):
            raise ValueError("I don't think so -- values are %s, %s" %
                             (width, height))
        Blob.__init__(self, width, height,
                      color, emphasis, highlight)
```

Blank lines

Separate top-level function and class definitions with two blank lines.

Method definitions inside a class are separated by a single blank line.

Extra blank lines may be used (sparingly) to separate groups of related functions. Blank lines may be omitted between a bunch of related one-liners (for example, a set of dummy implementations).

Use blank lines in functions, sparingly, to indicate logical sections.

Python accepts the Control-L (^L) form feed character as whitespace. Many tools treat these characters as page separators, so you may use them to separate pages of related sections of your file.

Imports

Imports should usually be on separate lines, for example:

```
import os
import sys
```

Don't put them together like this:

```
import sys, os
```

It's okay to say this, though:

```
from subprocess import Popen, PIPE
```

Imports are always put at the top of the file, just after any module comments and docstrings and before module globals and constants.

Imports should be grouped in the following order:

1. Standard library imports
2. Related third-party imports
3. Local application/library–specific imports

Put a blank line between each group of imports.

Put any relevant `__all__` specification after the imports.

Relative imports for intra-package imports are highly discouraged. Always use the absolute package path for all imports. Even now that PEP 328[5] is fully implemented in Python 2.5, its style of explicit relative imports is actively discouraged; absolute imports are more portable and usually more readable.

When importing a class from a class-containing module, it's usually okay to spell them

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

If this spelling causes local name clashes, then spell them

```
import myclass
import foo.bar.yourclass
and use myclass.MyClass and foo.bar.yourclass.YourClass.
```

Whitespace in expressions and statements

Pet peeves—avoid extraneous whitespace in the following situations:

- Immediately inside parentheses, brackets, or braces

  Yes:

  ```
  spam(ham[1], {eggs: 2})
  ```

[5] PEP 328, Imports: Multi-Line and Absolute/Relative

No:

```
spam( ham[ 1 ], { eggs: 2 } )
```

- Immediately before a comma, semicolon, or colon

  Yes:

```
if x == 4: print x, y; x, y = y, x
```

  No:

```
if x == 4 : print x , y ; x , y = y , x
```

- Immediately before the open parenthesis that starts the argument list of a function call

  Yes:

```
spam(1)
```

  No:

```
spam (1)
```

- Immediately before the open parenthesis that starts an indexing or slicing

  Yes:

```
dict['key'] = list[index]
```

  No:

```
dict ['key'] = list [index]
```

- More than one space around an assignment (or other) operator to align it with another

  Yes:

```
x = 1
y = 2
long_variable = 3
```

  No:

```
x             = 1
y             = 2
long_variable = 3
```

OTHER RECOMMENDATIONS

Always surround these binary operators with a single space on either side: assignment (`=`), augmented assignment (`+=`, `-=`, and so on), comparisons (`==`, `<`, `>`, `!=`, `<>`, `<=`, `>=`, `in`, `not in`, `is`, `is not`), and Booleans (`and`, `or`, `not`).

Use spaces around arithmetic operators.

Yes:

```
i = i + 1
submitted += 1
```

```
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

No:

```
i=i+1
submitted +=1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

Don't use spaces around the = sign when used to indicate a keyword argument or a default parameter value.

Yes:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

No:

```
def complex(real, imag = 0.0):
    return magic(r = real, i = imag)
```

Compound statements (multiple statements on the same line) are generally discouraged.

Yes:

```
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

While sometimes it's okay to put an `if/for/while` with a small body on the same line, never do this for multiclause statements. Also avoid folding such long lines!

Rather not:

```
if foo == 'blah': do_blah_thing()
for x in lst: total += x
   while t < 10: t = delay()
```

Definitely not:

```
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()
try: something()
finally: cleanup()
do_one(); do_two(); do_three(long, argument,
                             list, like, this)
if foo == 'blah': one(); two(); three()
```

Comments

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up to date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it's an identifier that begins with a lowercase letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted. Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

You should use two spaces after a sentence-ending period.

When writing English, Strunk and White apply.

Python coders from non-English-speaking countries: please write your comments in English, unless you are 120% sure that the code will never be read by people who don't speak your language.

Block comments

Block comments generally apply to some (or all) code that follows them and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Inline Comments

Use inline comments sparingly.

An inline comment is a comment on the same line as a statement. Inline comments should be separated by at least two spaces from the statement. They should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

Documentation strings

Conventions for writing good documentation strings (aka docstrings) are immortalized in PEP 257.[6]

Write docstrings for all public modules, functions, classes, and methods. Docstrings are not necessary for nonpublic methods, but you should have a comment that describes what the method does. This comment should appear after the `def` line.

PEP 257 describes good docstring conventions. Note that, most importantly, the `"""` that ends a multiline docstring should be on a line by itself and preferably preceded by a blank line, for example:

```
"""Return a foobang
```

[6] PEP 257, Docstring Conventions, Goodger, van Rossum

```
Optional plotz says to frobnicate the bizbaz first.

"""
```

For one-liner docstrings, it's okay to keep the closing `"""` on the same line.

VERSION BOOKKEEPING

If you have to have Subversion, CVS, or RCS crud in your source file, do it as follows:

```
    __version__ = "$Revision: 68852 $"     # $Source$
```

These lines should be included after the module's docstring, before any other code, separated by a blank line above and below.

Naming conventions

The naming conventions of Python's library are a bit of a mess, so we'll never get this completely consistent. Nevertheless, here are the currently recommended naming standards. New modules and packages (including third-party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

DESCRIPTIVE: NAMING STYLES

There are many different naming styles. It helps to be able to recognize what naming style is being used, independent of what it's used for.

The following naming styles are commonly distinguished:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (or CapWords, or CamelCase—so named because of the bumpy look of its letters[7]). This is also sometimes known as StudlyCaps.
  Note: When using abbreviations in CapWords, capitalize all the letters of the abbreviation. Thus `HTTPServerError` is better than `HttpServerError`.
- mixedCase (differs from CapitalizedWords by initial lowercase character!)
- Capitalized_Words_With_Underscores (ugly!)

There's also the style of using a short unique prefix to group related names together. This is seldom used in Python, but I mention it for completeness. For example, the `os.stat()` function returns a tuple whose items traditionally have names like `st_mode`, `st_size`, `st_mtime`, and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

---

[7] www.wikipedia.com/wiki/CamelCase

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

- _single_leading_underscore

  Weak "internal use" indicator. For example, `from M import *` does not import objects whose name starts with an underscore.

- single_trailing_underscore_

  Used by convention to avoid conflicts with Python keyword. For example, `tkinter.Toplevel(master, class_='ClassName')`.

- __double_leading_underscore

  When naming a class attribute, it invokes name mangling (inside class `FooBar`, `__boo` becomes `_FooBar__boo`; see below).

- __double_leading_and_trailing_underscore__

  "Magic" objects or attributes that live in user-controlled namespaces. For example, `__init__`, `__import__` or `__file__`. Never invent such names; use them only as documented.

PRESCRIPTIVE: NAMING CONVENTIONS

- Names to avoid

  Never use the characters *l* (lowercase letter el), *O* (uppercase letter oh), or *I* (uppercase letter eye) as single-character variable names.

  In some fonts, these characters are indistinguishable from the numerals 1 (one) and 0 (zero). When tempted to use *l*, use *L* instead.

- Package and module names

  Modules should have short, all-lowercase names. Underscores can be used in a module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

  Since module names are mapped to filenames, and some file systems are case insensitive and truncate long names, it's important that module names be fairly short—this won't be a problem on UNIX, but it may be a problem when the code is transported to older Mac or Windows versions or DOS.

  When an extension module written in C or C++ has an accompanying Python module that provides a higher-level (for example, more object-oriented) interface, the C/C++ module has a leading underscore (for example, `_socket`).

- Class names

  Almost without exception, class names use the CapWords convention. Classes for internal use have a leading underscore in addition.

- Exception names

Because exceptions should be classes, the class-naming convention applies here. However, you should use the suffix `Error` on your exception names (if the exception actually is an error).

- Global variable names

  (Let's hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

  Modules that are designed for use via `from M import *` should use the `__all__` mechanism to prevent exporting globals or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are module nonpublic).

- Function names

  Function names should be lowercase, with words separated by underscores as necessary to improve readability.

  mixedCase is allowed only in contexts where that's already the prevailing style (for example, threading.py), to retain backward compatibility.

- Function and method arguments

  Always use `self` for the first argument to instance methods.

  Always use `cls` for the first argument to class methods.

  If a function argument's name clashes with a reserved keyword, it's generally better to append a single trailing underscore than to use an abbreviation or spelling corruption. Thus, `print_` is better than `prnt`. (Perhaps better is to avoid such clashes by using a synonym.)

- Method names and instance variables

  Use the function-naming rules: lowercase with words separated by underscores as necessary to improve readability.

  Use one leading underscore only for nonpublic methods and instance variables.

  To avoid name clashes with subclasses, use two leading underscores to invoke Python's name-mangling rules.

  Python mangles these names with the class name: if class `Foo` has an attribute named `__a`, it cannot be accessed by `Foo.__a`. (An insistent user could still gain access by calling `Foo._Foo__a`.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

  Note: there is some controversy about the use of __names (see below).

- Constants

  Constants are usually declared on a module level and written in all capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL`.

- Designing for inheritance

Always decide whether a class's methods and instance variables (collectively called attributes) should be public or nonpublic. If in doubt, choose nonpublic; it's easier to make it public later than to make a public attribute nonpublic.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backward-incompatible changes. Nonpublic attributes are those that are not intended to be used by third parties; you make no guarantees that nonpublic attributes won't change or even be removed.

We don't use the term *private* here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes includes those that are part of the subclass API (often called *protected* in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class's behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

- Public attributes should have no leading underscores.
- If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, `cls` is the preferred spelling for any variable or argument that's known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

- For simple public data attributes, it's best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Properties work only on new-style classes.

Note 2: Try to keep the functional behavior side-effect free, although side effects such as caching are generally fine.

Note 3: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

- If your class is intended to be subclassed, and you have attributes that you don't want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python's name-mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and `__getattr__()`, less convenient. However the name-mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

Programming recommendations

You should write code in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Pyrex, Psyco, and such).

For example, don't rely on CPython's efficient implementation of in-place string concatenation for statements in the form `a+=b` or `a=a+b`. Those statements run more slowly in Jython. In performance-sensitive parts of the library, you should use the `''.join()` form instead. This will ensure that concatenation occurs in linear time across various implementations.

Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators.

Also, beware of writing `if x` when you really mean `if x is not None`, for example, when testing whether a variable or argument that defaults to `None` was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

Use class-based exceptions.

String exceptions in new code are forbidden, because this language feature has been removed in Python 2.6.

Modules or packages should define their own domain-specific base exception class, which should be subclassed from the built-in `Exception` class. Always include a class docstring, for example:

```
class MessageError(Exception):
    """Base class for errors in the email package."""
```

Class-naming conventions apply here, although you should add the suffix `Error` to your exception classes if the exception is an error. Non-error exceptions need no special suffix.

When raising an exception, use `raise ValueError('message')` instead of the older form `raise ValueError, 'message'`.

The paren-using form is preferred because when the exception arguments are long or include string formatting, you don't need to use line continuation characters thanks to the containing parentheses. The older form has been removed in Python 3.

When catching exceptions, mention specific exceptions whenever possible instead of using a bare `except:` clause. For example, use

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

A bare `except:` clause will catch `SystemExit` and `KeyboardInterrupt` exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use `except Exception:`.

A good rule of thumb is to limit use of bare `except` clauses to two cases:

- If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.
- If the code needs to do some cleanup work but then lets the exception propagate upward with `raise`, then `try...finally` is a better way to handle this case.

In addition, for all `try/except` clauses, limit the `try` clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs.

Yes:

```
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
```

No:

```
try:            # Too broad!
    return handle_value(collection[key])
except KeyError:
    return key_not_found(key)
```

Will also catch `KeyError` raised by `handle_value()`

Use string methods instead of the string module.

String methods are always much faster and share the same API with Unicode strings. Override this rule if backward compatibility with Python versions older than 2.0 is required.

Use `''.startswith()` and `''.endswith()` instead of string slicing to check for prefixes or suffixes.

`startswith()` and `endswith()` are cleaner and less error prone.

Yes:

```
if foo.startswith('bar'):
```

No:

```
if foo[:3] == 'bar':
```

The exception is if your code must work with Python 1.5.2 (but let's hope not!).

Object type comparisons should always use `isinstance()` instead of comparing types directly.

Yes:

```
if isinstance(obj, int):
```

No:

```
if type(obj) is type(1):
```

When checking to see if an object is a string, keep in mind that it might be a Unicode string too! In Python 2.3, `str` and `unicode` have a common base class, `basestring`, so you can do the following:

```
if isinstance(obj, basestring):
```

In Python 2.2, the `types` module has the `StringTypes` type defined for that purpose, for example:

```
from types import StringTypes
if isinstance(obj, StringTypes):
```

In Python 2.0 and 2.1, you should do the following:

```
from types import StringType, UnicodeType
if isinstance(obj, StringType) or \
   isinstance(obj, UnicodeType) :
```

For sequences (strings, lists, tuples), use the fact that empty sequences are false.

Yes:

```
if not seq:        if seq:
```

No:

```
if len(seq)       if not len(seq)
```

Don't write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable, and some editors (or more recently, reindent.py) will trim them.

Don't compare boolean values to `True` or `False` using `==`.

Yes:

```
if greeting:
```

No:

```
if greeting == True:
```

Worse:

```
if greeting is True:
```

Copyright—this document has been placed in the public domain.

The Zen of Python

The following document is PEP 20, also referred to as "The Zen of Python," a slightly tongue-in-cheek statement of the philosophy of Python. In addition to being included in the Python documentation, the Zen of Python is also an Easter egg in the Python interpreter. Type `import this` at the interactive prompt to see it.

Long time Pythoneer Tim Peters succinctly channels the BDFL's (Benevolent Dictator for Life) guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down.

The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!

index

Symbols

Numerics

A

D

G

H

I

J

K

L

Q

R

S

T

U

V

W

Z

PROGRAMMING

THE Quick Python Book SECOND EDITION

Vernon L. Ceder

This revision of Manning's popular **The Quick Python Book** offers a clear, crisp introduction to the elegant Python programming language and its famously easy-to-read syntax. Written for programmers new to Python, this updated edition covers features common to other languages concisely, while introducing Python's comprehensive standard functions library and unique features in detail.

After exploring Python's syntax, control flow, and basic data structures, the book shows how to create, test, and deploy full applications and larger code libraries. It addresses established Python features as well as the advanced object-oriented options available in Python 3. Along the way, you'll survey the current Python development landscape, including GUI programming, testing, database access, and web frameworks.

What's Inside

- Concepts and Python 3 features
- Regular expressions and testing
- Python tools
- All the Python you need—nothing you don't

Second edition author **Vern Ceder** is Director of Technology at the Canterbury School in Fort Wayne, Indiana where he teaches and uses Python. The first edition of this book was written by **Daryl Harms** and **Kenneth McDonald**.

For online access to the author, and a free ebook for owners of this book, go to manning.com/TheQuickPythonBookSecondEdition

Free ebook
SEE INSERT

"The quickest way to learn the basics of Python."
—Massimo Perga, Microsoft

"This is my favorite Python book... a competent way into serious Python programming."
—Edmon Begoli
Oak Ridge National Laboratory

"Great book... covers the new incarnation of Python."
—William Kahn-Greene
Participatory Culture Foundation

"Like Python itself, its emphasis is on readability and rapid development."
—David McWhirter, Cranberryink

"Python coders will love this nifty book."
—Sumit Pal, Leapfrogrx

ISBN 13: 978-1-935182-20-7
ISBN 10: 1-935182-20-X
53999
9781935182207

MANNING $39.99 / Can $49.99 [INCLUDING eBOOK]